TRABAJO FINAL DE CARRERA

TITULO: Implementación de un juego en red para móviles.

AUTOR: Francisco Javier López Solanes

DIRECTOR: Sergio Machado Sánchez

FECHA: 30 de junio de 2005

TITULO: Implementación de un juego en red para móviles**.**

AUTOR: Francisco Javier López Solanes

DIRECTOR: Sergio Machado Sánchez

FECHA: 30 de junio de 2005

Resumen

En el presente proyecto se realiza el diseño e implementación de un juego en red, multijugador, para dispositivos móviles utilizando el lenguaje de programación Java 2 Micro Edition.

Para el desarrollo se han utilizado las herramientas que facilita la empresa Sun Microsystems y el entorno de desarrollo Eclipse, todos son programas de libre distribución, sin ser necesaria la obtención de licencias.

La aplicación desarrollada es un juego de rol en el que pueden participar hasta seis jugadores y el objetivo del cual consiste en eliminar a todos los jugadores rivales a lo largo de las diferentes batallas por turnos que tengan lugar durante la partida. Cada jugador puede elegir entre diferentes personajes. Según la elección, el jugador posee unas u otras características que deberá aprovechar para vencer a sus rivales.

Al tratarse de una aplicación en red, ha sido necesario el diseño de un protocolo de comunicación en arquitectura Cliente-Servidor. El protocolo de transporte escogido para dicho protocolo ha sido UDP. Se ha desarrollado, junto a la aplicación cliente, un servidor de juego que centraliza las comunicaciones de los clientes, para el desarrollo de dicho servidor se ha utilizado Java 2 Stardad Edition.

El juego presenta una interfaz de usuario de bajo nivel, para el desarrollo del cual se han utilizado diferentes imágenes y gráficos de libre distribución para no tener que diseñarlos, puesto que esto quedaba fuera del alcance de este proyecto.

Debido a las limitaciones propias de los dispositivos móviles, se ha intentado optimizar al máximo la utilización de recursos tanto de procesador como de memoria escogiendo un formato de imagen (PNG) que reduce, frente a otros formatos, el tamaño que ocupa en memoria cada una de estas imágenes y optando por la programación de más bajo nivel que permite el lenguaje elegido.

Title: Implementation of a network game for mobile devices.

Author: Francisco Javier López Solanes

Director: Sergio Machado Sánchez

Date: June, 30th 2005

Overview

In the present project there is realized the design and implementation of a network multiplayer game, for mobile devices using the programming language Java 2 Micro Edition.

For the development there have been in use the toolkits of Sun Microsystems company and the development environment Eclipse, they all are free distribution programs, without being necessary the obtaining license.

The developed application is a role game in which they can take part up to six players, and the aim of which it consists in the elimination of all the rival players along the different turn based battles that take place during the game. Every player can choose between different heros. According to the election, the player have some or other features that he will have to exploit to beat his opponents.

In the way that the aplication is network based, there has been necessary the design of a communication protocol in a Server-Client architecture. UDP is the transport protocol choosed for de desing of communication protocol. There has developed, close to the application client, a game server who centralizes the communications of the clients, in the development of this server, Java 2 Stardad Edition is the programming language used for.

The game one presents a low-level user interface, for the development of which there have been in use different images and graphs of free distribution in the way to not having to design them, because it is out of the range of this Project.

Due to the limitations of the mobile devices, has been tried to optimize to the maximum the utilization of resources both of processor and of memory choosing an image format (PNG) that reduces, opposite to other formats, the size of used memory by each of these images and choosing for the programming the lower level that allows the chosen language.

A todos los que siempre habéis confiado en que éste momento llegaría, a los que me habéis enseñado que el camino solo es largo si uno piensa únicamente en la meta y no disfruta del viaje, a todos los compañeros y amigos que habéis hecho, de la que finaliza con este proyecto, una etapa inolvidable en mi vida y en especial a mis padres por estar siempre ahí.

Sólo puedo daros las gracias a todos. Sin vosotros nunca habría conseguido escribir estas líneas, GRÁCIAS.

ÍNDICE

INTRODUCCIÓN

Actualmente las telecomunicaciones y en especial aquellas tecnologías basadas en la movilidad se han instaurado en la vida cotidiana de prácticamente todos los ciudadanos de países occidentales y de una forma más acentuada en ciertos países asiáticos. Desde el inicio de la que podríamos llamar era tecnológica, se han utilizado todos los avances a nuestra disposición en el desarrollo de soluciones de negocio y empresa aunque de forma paralela este avance se ha vinculado al ocio y el tiempo libre, desde que Willy Higgimbothan en 1958, desarrollara "Tennis for two" la informática primero, internet después y la tecnología móvil actualmente han jugado y están jugando un papel fundamental en aportar soluciones de entretenimiento que ocupan gran parte de nuestro tiempo libre.

Por esto parecía adecuado sino imprescindible presentar un proyecto académico que se acerque al mundo de los videojuegos y lo hiciera tratando de aprovechar la tecnología más actual en medida de lo posible, cosa que ha desembocado en el desarrollo de un juego diseñado para dispositivos móviles.

GPRS y UMTS más recientemente son tecnologías que permiten a nuestros terminales móviles conectarse a redes IP como internet, la más extensa y conocida por todos. Es por esto que parecía necesario aprovechar dichas tecnologías para desarrollar un juego no solo orientado a dispositivos móviles si no que además permitiera jugar en red a varios jugadores.

CAPITULO 1. PRESENTACIÓN DEL PROYECTO

1.1. Objetivos del TFC:

El objetivo de éste proyecto es estudiar de forma académica, sin entrar en profundidad en la evaluación de las limitaciones tecnológicas que encontraríamos en un entorno real y de negocio (retardo, problema de sincronismo, público objetivo, penetración de mercado…), el diseño y desarrollo de un juego multijugador orientado a dispositivos móviles. El proyecto se ha basado en el estudio y aprendizaje de la versión para equipos wireless del lenguaje de programación Java, así como diseñar y desarrollar un protocolo de comunicación basado en arquitectura cliente-servidor que permita la interacción de diversos terminales con un servidor de juego que se encarga a su vez de comunicar de forma indirecta a estos terminales entre si.

1.2. Especificaciones iniciales:

Las especificaciones iniciales del juego que se debía desarrollar eran las de conseguir implementar un juego de rol multijugador. El escenario seria un mundo formado por 6 zonas de juego, 4 de las cuales eran habitaciones unidas

entre ellas por un pasillo central que ocupa el espacio de 2 habitaciones (dos pantallas).

Los jugadores podrían formar equipos y enfrentarse entre ellos en batallas por turnos, en cada turno el jugador podría atacar a un rival, lanzar un hechizo o curar a un compañero.

El ganador del juego sería el jugador que se mantenga en vida el resto irían pereciendo en las diferentes batallas en las que participe. Cualquier jugador que tome parte en una batalla de soportar al menos un turno de cada jugador rival pudiendo después abandonar el escenario de la batalla i huir.

1.3. ESTIRPE : Un juego en red para móviles.

ESTIRPE nos presenta un mundo fantástico en el que diferentes clanes luchan entre ellos por obtener el poder sobre el mundo de Fantasía. En una época en que las armas y la magia son el único medio para imponer el poder de nuestro clan sobre el resto de grupos y dominar así Fantasía.

Figura 1.1. Imagen de presentación del juego.

El jugador podrá escoger entre cinco héroes distintos para conseguir eliminar a los otros jugadores antes de caer. Cada uno de estos personajes posee diferentes habilidades y cualidades respecto al resto de personajes, la correcta utilización de dichas habilidades en la batalla marcará la diferencia entre la derrota y la victoria.

1.3.1. Personajes:

-  Mago: No posee grandes habilidades en la lucha cuerpo a cuerpo y es poco resistente a los ataques del resto de héroes, en contrapartida es el personaje que mayor puntos de magia posee al inicio del juego, por lo que si se sabe administrar dichos puntos podrá vencer a sus adversarios invocando hechizos de ataque o curación en los momentos claves de la batalla.

-  Guerrero: Es el personaje que mayor partido saca de su poder de ataque en el combate cuerpo a cuerpo. Se caracteriza por una resistencia media a ataques y una muy baja capacidad de realizar hechizos de magia, siendo el héroe que con menos puntos de magia inicia la aventura.

-  Amazona: La representación femenina en la lucha por el poder, definida por el equilibrio en sus habilidades. No posee fuerza ni resistencia extraordinaria aunque puede sacar ventaja de las diferentes armas empleadas en combate cuerpo a cuerpo y empieza el juego con un buen número de puntos de magia.

-  Enano: La rudeza es su mayor característica, el más resistente de los héroes, duro con buenas habilidades en ataque con armas, sin llegar al nivel de ataque del guerrero compensa su poco nivel de ataques de magia con una extraordinaria capacidad de encajar todos los envites y sortilegios que puedan ser lanzados sobre el.

-  Arquero: Junto con la amazona el jugador más equilibrado en cuanto a sus habilidades de defensa y ataque y su capacidad por realizar hechizos de magia. Cuenta con una resistencia media y un buen número de puntos de magia al comienzo de cada partida.

1.3.2. Indicadores:

Durante todo el transcurso del juego el estado de nuestro personaje, sus puntos de vida, puntos de magia, de ataque y defensa, se muestran al jugador por medio de varios indicadores.

Los puntos de vida y magia son representados por dos barras situadas en la parte inferior de la pantalla, que disminuyen a medida que el jugador pierde o gasta dichos puntos. La barra de color verde representa los puntos de vida y la de color azul los de magia. A ambos lados de las barras tenemos dos iconos

que representan mediante un valor numérico tanto los puntos de ataque como los de defensa.

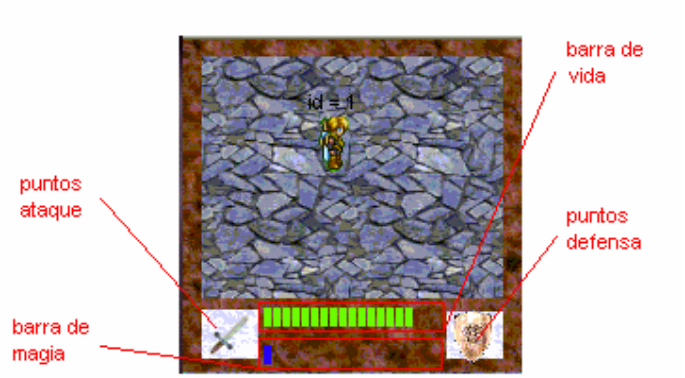

Figura 1.2. Indicadores del juego

1.4. Modo Historia:

En el inicio del juego, todos los jugadores se encuentran en modo historia y se pueden mover por las distintas zonas del escenario, interactuando con el resto de jugadores, uniéndose a ellos para formar un grupo de alianza o solicitar el inicio de una batalla que los enfrente.

Cuando un jugador solicita unirse a otro, deja de tener el control sobre su personaje y éste pasa a seguir los movimientos del jugador al que se ha unido entrando en batalla cuando éste lo haga. Un jugador unido a otro puede liberarse de la unión cuando lo desee o si el jugador al que se ha unido muere o abandona una batalla.

1.5. Modo Batalla:

Cuando dos o más jugadores se enzarzan en una batalla, éstos son trasladados a un escenario especial, donde se producen todos los enfrentamientos. Se establecerán turnos de batalla, el jugador que ha solicitado la batalla con el resto será el poseedor del primer turno de ataque/defensa y el servidor establecerá el orden de turnos del resto de participantes en la batalla. Una vez todos los jugadores han lanzado el primer ataque, cualquiera de ellos puede abandonar la batalla y huir. La batalla termina cuando todos los participantes menos uno han abandonado o han perecido en el transcurso del enfrentamiento.

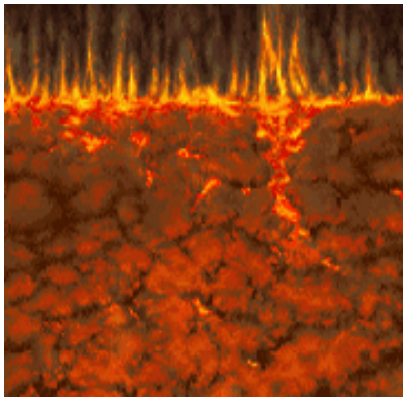

Figura 1.3. Escenario de Batalla.

Cuando un jugador ataca a otro, el jugador atacante se desplaza hasta la posición del jugador objetivo, lanza su ataque y regresa a su posición inicial a la espera de ser atacado por cualquiera de los jugadores que participan en la batalla.
En las siguientes imágenes se muestran varias imágenes del transcurso de una batalla.

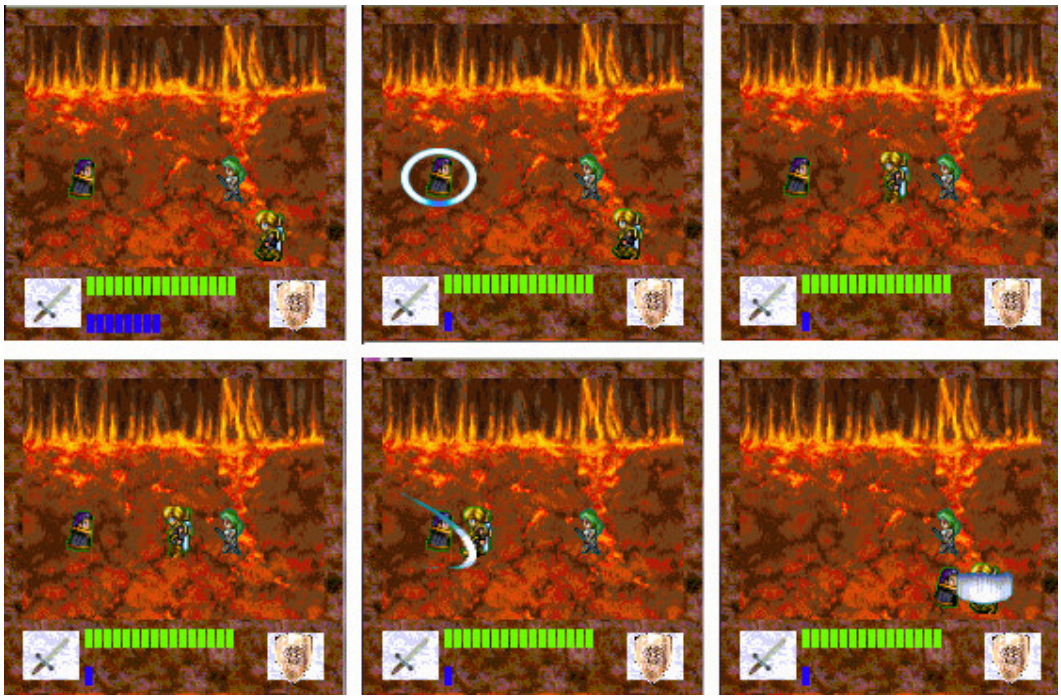

Figura 1.4. Secuencia de imágenes durante una batalla.

Si un jugador se encuentra en modo de batalla, en el escenario del mundo de Fantasía, se le identificará con una marca de batalla y el resto de jugadores

que permanezcan en modo historia no podrán interactuar con él hasta que regrese de la batalla.

CAPITULO 2. LENGUAJE UTILIZADO (J2ME)

El lenguaje utilizado para el desarrollo del proyecto ha sido el conocido lenguaje de programación, Java en su versión para dispositivos móviles (Java 2 Micro Edition). Esta versión ha sido adaptada a las limitaciones en los recursos tales como poca potencia de procesado, baja capacidad de memoria o interfaz de usuario reducida que ofrecen teléfonos móviles, agendas electrónicas y otros dispositivos móviles.

2.1. Perfil, Configuración y Máquina Virtual

Existen dos tipos de configuraciones para J2ME, CLDC Y CDC, Connected Limited Device Configuration y Connected Device Configuration respectivamente, la primera orientada a dispositivos con conexión y recursos limitados en cuanto a interfaz gráfica y capacidad de procesamiento de memoria, como teléfonos móviles y agendas electrónicas. La segunda configuración va dirigida a dispositivos con menos limitaciones, como sistemas de navegación o televisiones con internet. Cada configuración define las clases mínimas que los terminales pueden usar, así como los requisitos mínimos que debe cumplir el intérprete de Java, que traduce a código máquina el código de las clases Java, es decir, la máquina virtual.

En el caso que nos ocupa, desarrollar una aplicación para un teléfono móvil, la configuración necesaria será CLDC y dicha configuración define KVM (Kilobyte Virtual Machine) como máquina virtual requerida. KVM es la máquina virtual más pequeña desarrollada por Sun, admite una carga de memoria entre 40Kb y 80 Kb y está preparada para microprocesadores de 16 y 32 bits, en las primeras versiones de CLDC, no admitía operaciones en coma flotante debido a las restricciones en las clases soportadas.

La configuración CLDC, obtiene 3 clases de la versión J2SE de Java e incluye una nueva clase. java.lang.*, java.io.* y java.util.* son las clase obtenidas de J2SE y java.microedition.io.* es la nueva clase que se incluye en la configuración.

Un perfil define un conjunto de características que identifican a un determinado grupo de dispositivos junto con las APIs que controlan aspectos propios de dichos dispositivos, como puedan ser el control del ciclo de vida de la aplicación, la gestión de eventos o las interfaces de usuario.

Se puede decir que un perfil es la capa más próxima al usuario y al desarrollador, las aplicaciones corren sobre un determinado perfil que a su vez está definido por la configuración. Existen varios perfiles para cada configuración y un mismo dispositivo puede soportar varios perfiles distintos.

MIDP es el primer perfil creado para J2ME y se trata de un perfil construido sobre la configuración CLDC, esta orientado a máquinas con escasos recursos en cuanto a capacidad de procesado y de memoria. Se trata del perfil sobre el que correrá la aplicación desarrollada para este proyecto.

Igual que la configuración CLDC, el perfil MIDP aporta un conjunto de clases necesarias para desarrollar las funciones propias de control sobre un determinado ambiente de ejecución. La principales clases aportadas por el perfil MIDP son:

-javax.microedition.midlet
-javax.microedition.lcdui
-javax.microedition.rms
-javax.microedition.io
-java.lang
-java.util

Se define MIDlet como la aplicación desarrollada específicamente para ejecutarse sobre el perfil MIDP.

2.2. ¿Por qué J2ME?:

Existen otros lenguajes de programación para dispositivos móviles, a parte de J2ME, basados o desarrollados a partir de versiones para computadoras como C++, MobileVB o OPL orientados a una plataforma específica, en la mayoría de los casos Symbian o EPOCS (SO a partir del cual surge Symbian) aunque existen otros sistemas como Palm OS o Pocket PC que pueden ser objetivo de aplicaciones desarrolladas en estos lenguajes. Existen grandes diferencias en la filosofía o manera de programar cada en cada uno de estos lenguajes. La programación de aplicaciones en C++, requiere que éstas sean compiladas en cada máquina que las ejecute, requiere mayor esfuerzo y el tiempo de aprendizaje es mayor si no se tiene experiencia en programación en C. OPL es un lenguaje interpretado, en cierto modo como Java, es decir necesita de un traductor o intérprete que transforme el código programado (comprensible para el humano) a código máquina (comprensible para el dispositivo que ejecuta la aplicación) el traductor de OPL es OPLTRANS. OPL ha sido impulsado y soportado por sourceforge (www.sourceforge.net), asociación dedicada ala programación de pequeños dispositivos que defiende las bases de la distribución libre (GPL). MobileVB está basado en la plataforma de desarrollo Visual Studio, integrando complementos y herramientas de Visual Basic 6.0. Este software ha sido desarrollado por Appforge (www.appforge.com).

J2ME es sin duda el lenguaje más utilizado para el desarrollo de juegos para dispositivos móviles actualmente y su implementación por parte de los fabricantes en los dispositivos es casi de obligado cumplimiento, debido a la gran demanda del mercado. J2ME es por otra parte un lenguaje más sencillo de programar que C++ siempre que se tengan conocimientos de programación orientada a objetos. La elección de J2ME reduce en gran parte el tiempo de

aprendizaje y ofrece unas prestaciones aceptables en cuanto al aprovechamiento de recursos se refiere, sin llegar a las que pueda ofrecernos C++, por supuesto.

2.3. ¿Por qué MIDP 2.0 Y CLDC 1.1?

Para el desarrollo de este proyecto se ha optado por la última versión tanto en la elección de la configuración como del perfil sobre el cual va a ejecutarse nuestra aplicación. Está elección está justificada por tratarse de un desarrollo en un entorno puramente académico, puesto que tanto MIDP 2.0 como CLDC 1.1 requieren mayores prestaciones de los equipos sobre los que vayan a trabajar.

En un escenario orientado a las necesidades del mercado y a las diferentes opciones de negocio hubiese sido preferible optar por las versiones anteriores de configuración y perfil para abarcar un mayor mercado, puesto que muchos de los equipos que hoy en día poseen los posibles usuarios, soportan únicamente MIDP 1.0 o CLDC 1.0 debido a sus limitaciones ya comentadas con anterioridad.

La elección de MIDP 2.0 y CLDC1.1, permite el uso de datagramas UDP para las comunicaciones y operaciones en coma flotante, aunque este último aspecto no vaya a jugar un papel tan decisivo en nuestra aplicación como el primero.

2.4. MIDlets:

Se de fine como MIDlet la aplicación desarrollada para el perfil MIDP y configuración CLDC. J2ME incluye el paquete javax.microedition.midlet.*, que define el comportamiento de una aplicación ante el entorno de ejecución. Este paquete incluye dos clases principales, MIDlet y MIDletstateChangeException, la primera define la estructura básica de nuestra ejecución, mientras que la segunda recoge todas las excepciones o fallos en el cambio de estado.

La clase MIDlet nos obliga a implementar tres métodos públicos, necesarios para la ejecución de la aplicación:

startApp(): es el primer método al que se llama cuando se inicia la ejecución de la aplicación o cuando ésta se reanuda.

pauseApp():es el método que se invoca cuando la aplicación debe ser detenida, normalmente por la recepción de una llamada entrante en el dispositivo móvil y debe definir las acciones que se deben llevar a cabo antes de pausar la ejecución.

destroyApp(): método invocado al finalizar la aplicación, se encarga de liberar y destruir todos los recursos y componentes utilizados durante la ejecución del MIDlet.

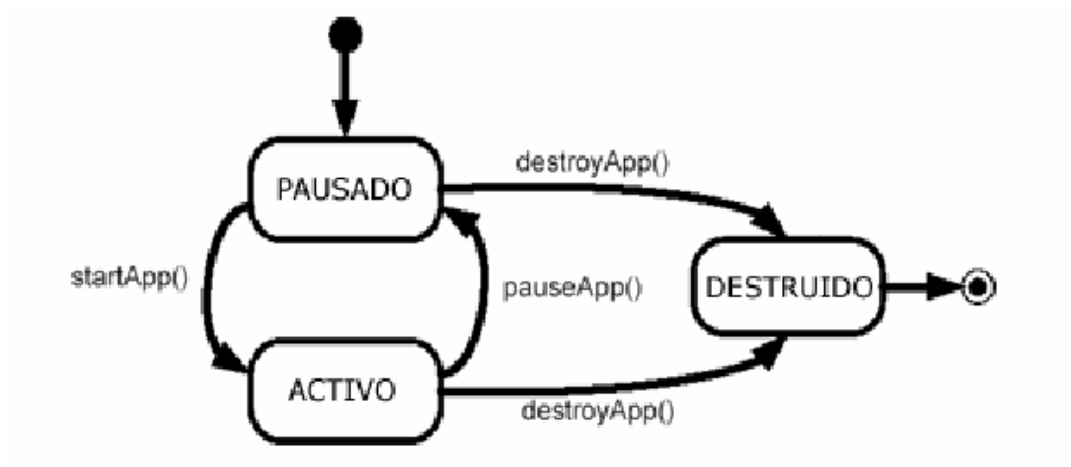

Figura 2.1. Ciclo de vida de un MIDlet.

CAPITULO 3. PROTOCOLO DISEÑADO

3.1. Arquitectura Cliente-Servidor

En muchas de las aplicaciones que implementan esta arquitectura la elección se basa en término de repartición de carga de procesado entre cliente y servidor, de esta forma se puede dar servicio a un mayor número de clientes a la vez puesto que el cliente ejecuta parte de la aplicación y el resto el servidor, es típica esta arquitectura en aplicaciones que atacan bases de datos. El servidor de bases de datos atiende a las peticiones de los diferentes clientes, pero éstos últimos son los encargados de controlar la interfaz de usuario, de almacenar y/o procesar los datos recibidos del servidor. En internet es típica la utilización de esta arquitectura, los servidores web atienden a las peticiones de información de los diversos clientes, pero son éstos los que ejecutan los programas navegadores.

En el caso del proyecto que se presenta el motivo fundamental de dicha elección se basa ya no en términos de repartición de carga de procesado, si no en la centralización de las comunicaciones entre clientes. Es el servidor el que aglutina toda la información transmitida por los clientes, la procesa y en función del resultado obtenido genera y transmite al los clientes que corresponda la nueva información. No es posible por tanto la comunicación directa entre clientes.

En la figura siguiente se muestra la arquitectura cliente-servidor utilizada en el desarrollo del juego

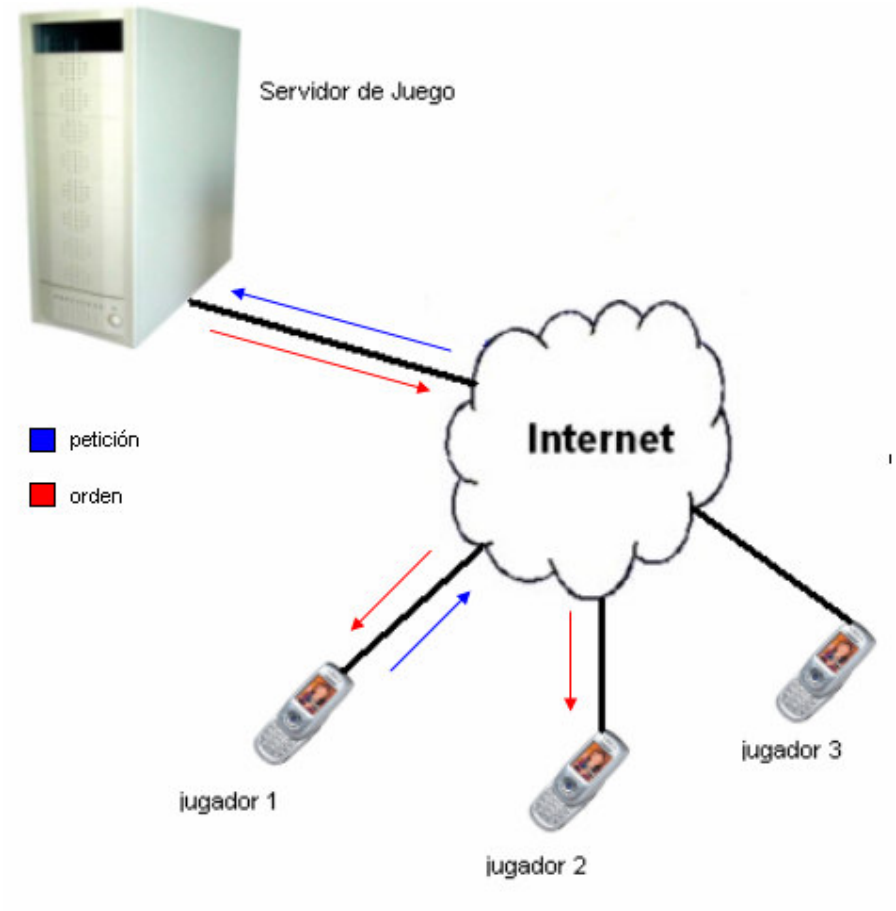

Figura 3.1. Arquitectura Cliente-Servidor

El protocolo diseñado se basa en el envío de peticiones por parte del cliente y de órdenes por parte del servidor, pasando a ser el servidor el gestor de cada partida. Cuando el servidor recibe una petición por parte de un cliente, la procesa y en función del resultado envía una orden al conjunto de jugadores a los que afecta la petición.

3.2. Elección del protocolo de transporte.

Tanto el perfil como la configuración elegida en el desarrollo de la aplicación, permiten el uso de datagramas UDP (User Datagram Protocol) para la comunicación a través de sockets entre diferentes dispositivos. UDP ha sido el protocolo de la capa de transporte escogido para el desarrollo del protocolo de

comunicación de la aplicación. Puesto que se trata de un juego en tiempo real es necesario escoger un protocolo que introduzca el menor retardo posible en el envío de paquetes, UDP es un protocolo que no confirma la recepción de paquetes, por lo que no debe esperarse la confirmación del primer paquete antes del envío del segundo. El protocolo de comunicación a nivel de aplicación que se ha diseñado no prevé la confirmación de paquetes sino que se utiliza un sistema de petición del cliente y orden del servidor, es decir el cliente realiza una solicitud al servidor, éste la evalúa y la procesa y en función del resultado envía un paquete que contiene una orden a los clientes que sea necesario. Uno de los clientes puede recibir un paquete de orden del servidor sin que éste haya enviado ningún paquete de petición al servidor, por lo que no se puede hablar de un sistema de confirmación.

3.3. Formato de paquetes.

Para que cliente y servidor puedan comunicarse, se ha tenido que diseñar un protocolo de comunicación basado en el envío de información del estado del juego en ambos sentidos de la comunicación. La información se ha empaquetado en datagramas UDP, dicha información ha tenido que ser formateada de manera que tanto el cliente como el servidor sean capaces de interpretarla. El formato que se ha escogido para la creación de los paquetes de información consiste en la creación de una cadena de caracteres (String) a partir de los datos a transmitir, dados en enteros, cada entero que representa una variable de juego es separada por un símbolo “/” del siguiente entero. El final de la cadena que contiene la información se marca con un símbolo “//”. Una vez se ha creado el String que contiene el mensaje a transmitir se convierte en un flujo de bytes que será encapsulado en el campo de datos del datagrama UDP. Tanto la clase Comunicacio (ver anexo 1) del cliente como la clase Server del servidor implementan un método que permite la lectura de caracteres separados por símbolos “/”, finalizando la lectura cuando encuentra dos símbolos de separación de datos seguidos.

A continuación se muestran los diferentes paquetes que forman el protocolo de comunicación entre cliente y servidor.

3.3.1. Paquetes de Alta:

Petición:

1	X_0	Y_0	Hab_0	Id_0	personaje	Defensa	Ataque	Magia

El paquete de petición de alta que envía el cliente contiene los mismos campos que un paquete de acción, a diferencia del número que forma la cabecera, los

primeros datos de cada paquete forman una cabecera que indica el tipo de paquete. En el caso de los paquetes del alta el identificador es “1”.

Personaje: Entero que identifica el Sprite (ver anexo 2)que debe cargarse cuando se desee representar al jugador.
Defensa, Ataque y Magia: Cada uno de estos campos representa los puntos de defensa, ataque y magia que posee el jugador, en función del personaje seleccionado.

Orden/Confirmación:

1	Id	Xini	Yini	Habini

Id: indica el identificador que el servidor ha asignado al cliente
Xini e Yini: indican la posición que ocupa en pantalla el jugador al iniciar la partida.
Habini: contiene el número de habitación en la que empieza el jugador.

3.3.2. Paquetes de Acción:

Petición:

2	X	Y	Hab	Id	personaje	Defensa	Ataque	Magia

Es el paquete que envía el cliente cuando el jugador modifica su posición, en el se indica la posición X e Y, la habitación que ocupa el personaje en este momento, su identificador, el personaje que lo representa y los puntos de defensa, ataque y magia que tiene el jugador.

Orden:

2	X	Y	Id	personaje

Cuando el servidor recibe un paquete de acción de un cliente, envía el paquete de orden de acción al resto de jugadores para que éstos actualicen la posición del jugador rival que ha modificado su posición y se les indica también cual es el personaje que representa a dicho jugador.

3.3.3. Paquetes de Unión:

Petición:

4	0/1	Id	Id

		Solicitante	Master

En la cabecera se indica que el paquete hace referencia a un paquete de unión puesto que el valor del primer campo es “4”, el valor del segundo campo varia entre “0” y “1” en función de si el paquete solicita la unión o la desunión a otro jugador respectivamente. El campo Id Solicitante hace referencia al identificador del jugador que realiza la petición y el campo Id Master al identificador del jugador al que desea unirse.

Orden:

4	0/1	Id Master

Este paquete se envía tanto al solicitante de la unión como al jugador al que se unirá el solicitante después de actualizar en el servidor el vector IdSlaves en el jugador que será jefe del grupo y actualizar el atributo IdMaster, en el jugador que actuará como esclavo, con la id del jefe de grupo.

En el caso que el segundo campo tenga el valor “1”, la solicitud es de desunión y también se envía a ambos jugadores.

3.3.4. Paquetes de Inicio de Batalla:

Petición:

3	0	Id Objetivo	Id Solicitante

En la cabecera se establece el número “3” como identificador de paquete de batalla para cada paquete que intervenga en este aspecto del juego, el “0” indica que se trata de un paquete de inicio de batalla.

Id Objetivo: especifica el identificador del jugador contra el que se desea iniciar una batalla. Si éste jugador tiene otro jugador asociado en alianza, éste también entrará en la batalla.

Id Solicitante: indica el identificador del jugador que solicita el inicio de la batalla.

Orden:

3	0	Num Jugadores	Id1	Id2	...	Idn

Es el paquete que recibe un cliente antes de pasar al modo batalla e instanciar el gestor de batalla e iniciarlo. Cuando el servidor recibe una petición de batalla, evalúa si el jugador objetivo o solicitante tienen establecida alguna alianza con otros jugadores y envía a todos el mismo paquete de orden de inicio de batalla.

Se trata del único paquete del juego que tiene una longitud variable, en función del número de jugadores que tomarán parte en la batalla.

NumJugadores: indica cuantos jugadores se enfrentarán en la batalla.

Id1, Id2,…: contienen el identificador de cada jugador implicado en la batalla.

3.3.5. Paquetes de Ataque:

Petición:

3	1	Id Objetivo	Id Local	Tipo Arma

Durante el modo de batalla se generan los paquetes de petición de ataque cada vez que uno de los jugadores ataca a otro, en este paquete se envía la id del jugador atacado en el campo Id Objetivo, la id del jugador que realiza la petición en el campo Id Local y el arma utilizada en el ataque en el campo Tipo Arma. La cabecera indica que el paquete corresponde al modo de batalla y hace referencia a un ataque.

Orden:

3	1	Id Objetivo	Id Atacante	Tipo Arma	HP	MP	Ataque	Defensa	Id Muerte

Cuando el servidor procesa un paquete de petición de batalla, envía a todos los participantes de la misma un paquete con la información necesaria para representar la animación del ataque en el cliente y actualizar la situación del jugador local.
Id Objetivo: identificador del jugador que recibe el ataque

Id Atacante: identificador del jugador que realiza el ataque

Tipo Arma: entero que indica al cliente el tipo de arma utilizada en el ataque que debe representar por pantalla.

HP, MP, Ataque y Defensa: estos campos hacen referencia a los valores con los que el jugador local debe actualizar su situación de juego.

Id Muerte: Este campo es utilizado para indicar a todos los jugadores cuando uno de los personajes abandona la batalla porque ha muerto. Si ninguno de los jugadores muere durante el ataque, el campo contiene un valor “-1”.

3.3.6. Paquetes de fin de batalla:

Petición y orden:

3	9	Id

En este caso el formato del paquete utilizado por el cliente para solicitar el abandono de la batalla y el que utiliza el servidor para indicar al resto de jugadores que uno de los participantes abandona es el mismo.

La cabecera indica referencia a modo de batalla y especifica que se trata de un paquete de fin de batalla. El campo Id contiene el identificador del jugador que abandona.

3.4. Entorno Ideal

En el diseño y la implementación del protocolo de comunicación del juego no se han tenido en cuenta diversos aspectos que no permitirían el correcto funcionamiento del juego en un entorno real. La red GPRS introduce grandes retardos en la recepción de paquetes que afectan a la latencia de cualquier comunicación a través de la red. Otro aspecto que se debería tener en cuenta en la adaptación del juego a un entorno real de comunicación es la pérdida de paquetes y la llegada desordenada de los mismos.

La aplicación desarrollada se ha ejecutado en un entorno ideal utilizando varios equipos conectados mediante LAN, el los que se ejecutan un servidor de juego y los clientes en el simulador del entorno de desarrollo.

CAPITULO 4. CAPITULO 4. ASPECTOS DE LA IMPLEMENTACIÓN DE UN JUEGO EN RED PARA MÓVILES

4.1. Gráficos

4.1.1. Entorno gráfico (interfaz de bajo nivel):

El perfil MIDP, define el paquete javax.microedition.lcdui para el control de la interfaz de usuario (LCD User Interface). Una interfaz de usuario define los elementos y componentes gráficos que mostrará el dispositivo por pantalla, existen dos interfases de usuario diferenciadas dentro de este paquete, la interfaz de alto nivel y la de bajo nivel. La interfaz de alto nivel proporciona al desarrollador objetos gráficos predefinidos como cajas de texto, listas, formularios, alertas,… El aspecto de estos objetos depende exclusivamente del dispositivo donde se ejecute nuestra aplicación, cosa que confiere una gran portabilidad a las aplicaciones basadas en interfaz de usuario de alto nivel. La utilización de de este conjunto de clases se limita habitualmente a aplicaciones de negocios, donde es necesaria la navegación por menús, las entradas de texto por teclado y la presentación de datos por pantalla.

La elección de una interfaz de bajo nivel proporciona al desarrollador un control total de los gráficos que se desean mostrar en pantalla. En la implementación de un juego, es necesario mostrar imágenes de fondo, gráficos en movimiento y diferentes indicadores que no pueden ser definidas por la interfaz de alto nivel, por lo que es prácticamente obligatorio decantarse por la interfaz de bajo nivel en la implementación de éste proyecto.

En el desarrollo de un MIDlet es necesario implementar al menos un objeto display, que es el encargado de gestionar y controlar la pantalla y los dispositivos de entrada. La clase displayable hereda de la clase display y representa las diferentes pantallas que se muestran por el LCD de nuestro dispositivo móvil. Un MIDlet puede definir tantos objetos displayable como se desee, pero solo se puede mostrar uno en pantalla. De la clase displayable heredan tanto la clase Screen como la clase Canvas, estas dos clases implementan la interfaz de alto y bajo nivel respectivamente.

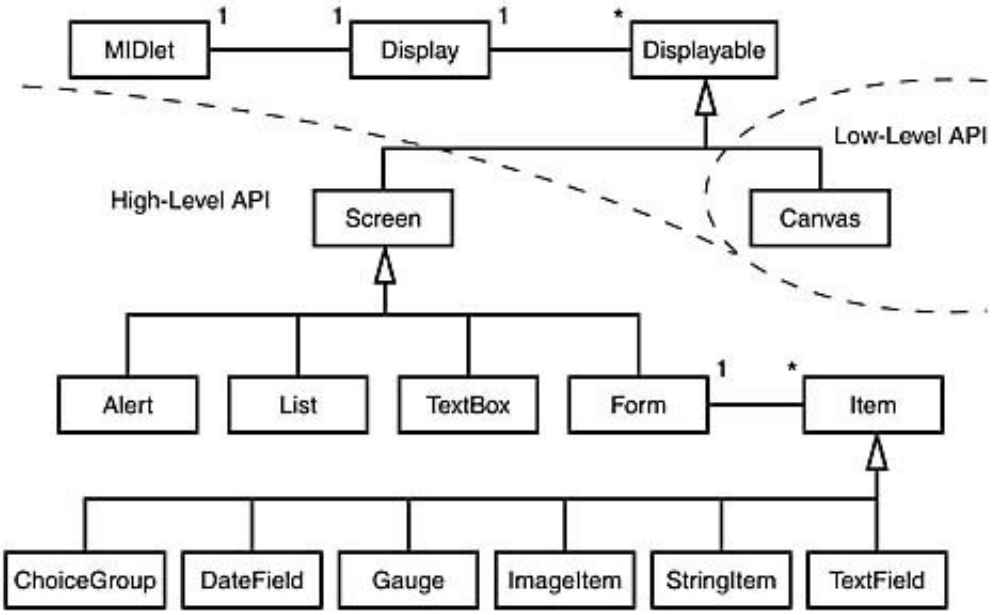

Figura 4.1. Jerarquía de clases a partir de Display

Para el desarrollo de éste proyecto y en base a lo comentado con anterioridad, se ha elegido la clase Canvas como la clase de la que hereda nuestra clase principal del juego SSCanvas, implementando por tanto la interfaz de bajo nivel. Desde la clase SSCanvas se controla la lectura del teclado y la renderización de los gráficos que se muestran en pantalla, es ésta la clase que implementa el método paint() (ver anexo 3), heredado de Canvas y encargado de pintar por pantalla. Ésta clase implementa a su vez el hilo principal del juego, se repite un bucle encargado de invocar a los métodos que controlan la lectura del teclado, la actualización del estado de los jugadores y las batallas y la renderización de gráficos. Desde SSCanvas se actualizan también los datos que se reporta desde el módulo de comunicaciones, la clase Comunicacio.

4.1.2. Formato de imagen:

Todas las imágenes y gráficos empleadas en el desarrollo del juego utilizan el formato de compresión PNG (Portable Network Graphics). Se trata de un formato de imagen que utiliza un algoritmo de compresión sin pérdida y libre de licencia. Éste formato permite llegar a los 48 bits por píxel y puede representar hasta 976 colores en la imagen. PNG permite especificar hasta 254 niveles de transparencia gracias al empleo del canal Alfa en sus capas, pero la mayor ventaja que aporta es que el tamaño de memoria que ocupa cada imagen en menor que en otros formatos, argumento más que suficiente para decantarse por PNG frente a otros formatos de imagen como GIF o JPEG en aplicaciones dirigidas a dispositivos móviles con reducido tamaño de memoria.

4.1.3. Personajes

La solución por la que se ha optado ante la dificultad de diseñar y digitalizar los gráficos que representarán a nuestros personajes, ha sido la de utilizar recursos de uso público dirigidos a programas de edición y desarrollo de juegos de rol como RPGMaker. Estos gráficos pueden descargarse de la red y existen diversos sitios que dedicados a recopilar, clasificar y ofrecer en descarga estos recursos. Los personajes se descargan en lienzos que contienen varias imágenes que representan distintas posiciones y orientaciones, en concreto para el desarrollo del juego se utilizan cuatro orientaciones posibles del jugador, norte, sur, este y oeste, para cada una de estas orientaciones, cada personaje posee tres posturas distintas. Alternando estas tres posturas o posiciones del personaje se consigue que parezca que éste está andando mientras avanza por la pantalla.

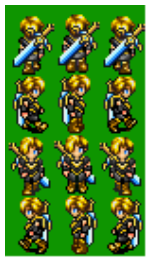

Figura 4.2. Lienzo del guerrero.

En la imagen de la figura anterior, se puede observar uno de los lienzos descargados de internet que se ha utilizado para representar el personaje del guerrero. Utilizando un programa de edición de imagen, más concretamente Gimp2 de libre distribución, se ha eliminado el fondo de color, dejando una capa transparente de fondo y se ha recortado cada frame almacenándolos como archivos independientes, posteriormente se han cargado las diferentes imágenes en un objeto Sprite que comentaremos más adelante.

Se ha escogido un tamaño estándar para cada personaje de 40x40 píxeles para que pueda apreciarse correctamente el personaje en la pantalla y sin que éste ocupe demasiado espacio en el mapa.

4.1.4. Mapas

De la misma manera que los personajes, las imágenes de los mapas se han descargado de sitios de internet que ofrecen recursos de libre distribución. Las imágenes descargadas, se han modificado y adaptado al tamaño de pantalla del dispositivo escogido para la implementación del juego, en este caso se han adaptado a la dimensiones del display del simulador que ofrece el kit de desarrollo de Sun, concretamente 177x180 píxeles.

Cada una de las pantalla o habitaciones que forman el mapa del juego se han cargado en un Sprite que alberga seis frames, uno por habitación. La selección de la habitación a mostrar en cada momento del juego depende de la habitación en la que se encuentra el jugador local. Cuando el personaje del jugador local abandona una habitación, el método ActualizaFondo (ver anexo 4) de la clase principal SSCanvas se encarga de modificar el frame seleccionado en función del costado por el cual el jugador abandona la habitación. Cuando se realiza una transición entre habitaciones, también debe corregirse la posición que ocupa en pantalla el jugador, para dar una sensación de continuidad. Por ejemplo si el jugador se encuentra en la habitación 1 y se mueve hacia la derecha y alcanza el límite de la habitación, se producirá una transición a la habitación 2, y la posición del jugador debe corregirse para que sea representado en el límite izquierdo de la pantalla en lugar del derecho que ocupaba hasta el momento.

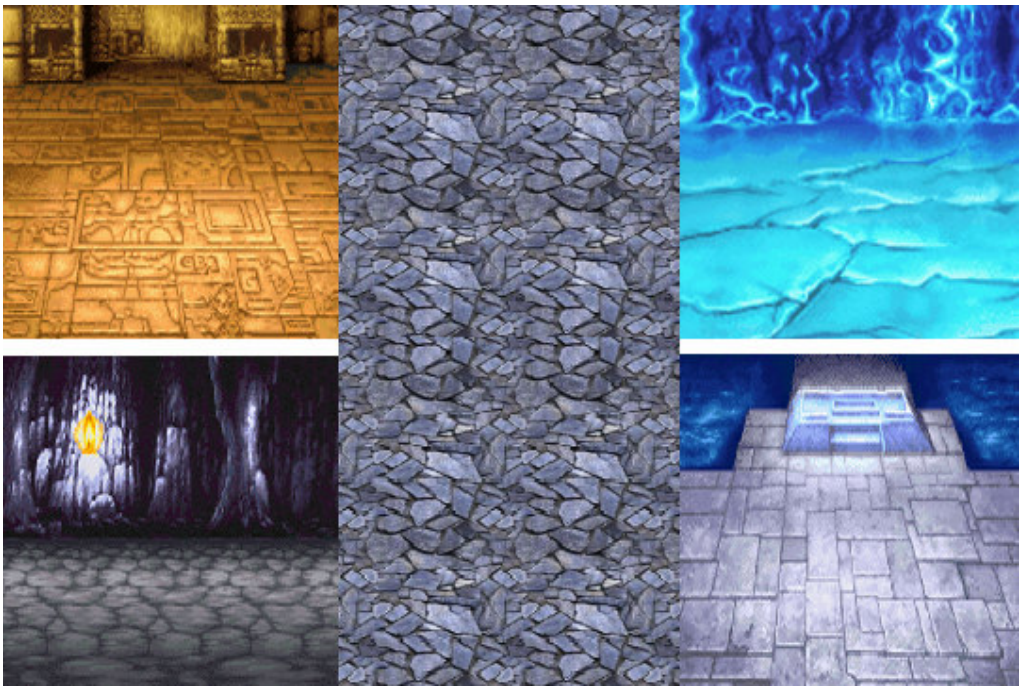

Figura 4.3. Composición del mapa de juego.

4.1.5. Clase Sprite

En la imagen siguiente se muestran de forma esquemática, en formato UML los atributos y métodos que implementa la clase Sprite:

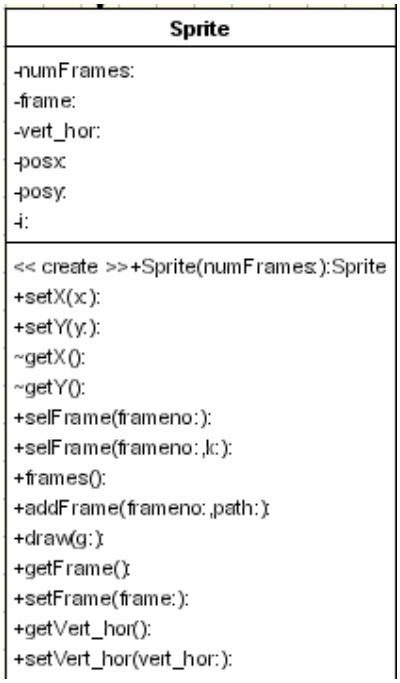

Figura 4.4. Diagrama UML de la clase Sprite.

Esta clase es la encargada de gestionar las imágenes que representan a cada personaje así como otros elementos gráficos que deben ser mostrados por pantalla. Cuando se inicia la ejecución de la aplicación, se cargan las imágenes en un vector que contiene cada uno de los frames que representa al personaje en cada una de sus posiciones o posturas mediante el método addFrame(). Se crea un objeto Sprite para cada personaje que hay que representar en pantalla, mediante los atributos de posición X e Y, se gestiona la zona del display en que se va a representar la imagen seleccionada. Para seleccionar la imagen o frame a mostrar en cada instante del juego se recurre al método selFrame(), éste método es invocado desde diferentes partes del código en función del gestor que controla la representación de cada imagen en cada momento. Por ejemplo el método ActualizarJugador de la clase principal SSCanvas invoca al método selFrame de la clase Sprite para escoger la posición o postura que debe mostrarse del personaje que representa al jugador local cuando éste se mueve, cambia de mapa o regresa del modo de batalla. Éste mismo método de la clase principal invoca a los métodos setX y setY para modificar la posición de pantalla en la que debe representarse la imagen del personaje.

Existen otras clases o módulos del programa que utilizan objetos del tipo Sprite para representar imágenes, como por ejemplo el gestor de batalla, los indicadores de vida y magia o el selector de objetivos.

El mapa de juego por ejemplo es almacenado en un solo objeto Sprite con seis frames que representan cada una de la habitaciones del mapa y en función de la habitación en la que se encuentra el jugador local se selecciona uno u otro frame para ser representado en el fondo de pantalla.

4.1.6. Clase Graphics

Para poder dibujar en pantalla imágenes, texto o figuras geométricas cuando la el objeto displayable escogido es canvas, debemos recurrir a crear un objeto de la clase Graphics. La clase Graphics nos ofrece varios métodos encargados de pintar en pantalla, el método paint de SSCanvas necesita que se le pase como argumento un objeto Graphics para poder dibujar en pantalla las distintas imágenes del juego.

4.1.7. Técnica del doble buffering

Cuando se dibujan diferentes imágenes en pantalla, corremos el riesgo de que estas parpadeen, creando un efecto nada agradable al usuario. Éste efecto se produce porque durante la ejecución de la aplicación el programa dibuja directamente en la pantalla las imágenes que se deben ir superponiendo para crear el fondo, el marco de indicadores y el personaje si la actualización de las imágenes a mostrar no se produce lo suficientemente rápido como para engañar al ojo humano. La técnica del doble buffering resuelve este problema, realizando la actualización o superposición de imágenes en un bloque de memoria o buffer, cuando la imagen a mostrar está completa, se vuelca directamente este bloque de memoria al buffer de la imagen final que se muestra por pantalla, con lo que el usuario no ve como se forma la composición de la imagen en el display, si no que ve la imagen completa en cada instante de refresco.

La elección de MIDP2 y CLCD1.1, facilitan la tarea del desarrollador permitiéndole olvidarse de implementar el doble buffering, puesto que el entorno de ejecución se encarga de forma transparente de “dibujar” en memoria lo que debe mostrarse posteriormente en pantalla, de este modo se ha podido realizar de forma secuencial la superposición de imágenes a mostrar dentro del método paint() de SSCanvas.

4.2. Lectura de teclado y Animaciones

La clase Canvas a parte de permitirnos dibujar por pantalla, ofrece la posibilidad de que el usuario interactúe con la aplicación mediante la utilización de tres métodos para la lectura de entrada del teclado llamados listeners (“escuchadores”), keyPressed(), keyReleased() y keyRepeated(). Cada uno de ellos recoge eventos diferentes de teclado, keyPressed() recoge la pulsación de una tecla, keyReleassed() evalúa cuando una tecla pulsada deja de estarlo y keyRepeated() controla la pulsación de la misma tecla repetidas veces. En la implementación del juego se ha optado por la utilización de dos de estos tres métodos heredados de Canvas por la clase SSCanvas.

4.2.1. Listeners (keyPressed() y keyReleased())

Cualquier acción realizada por el usuario sobre el teclado del dispositivo móvil, es recogida por uno de estos dos métodos. Cuando keyPressed() (ver anexo 5) o keyReleased() escuchan un evento, ejecutan el código que contiene el método.

En el juego el método keyPressed() de SSCanvas, controla únicamente el movimiento del jugador local aumentando o disminuyendo el incremento en la posición X o Y que deberá aplicar el método encargado de actualizar al jugador local. Por el contrario el método keyReleased realiza más de una acción en función del momento de la ejecución del juego en la que nos encontremos y de la tecla que el usuario deja de presionar, por ejemplo, si nos encontramos en modo historia podemos gestionar el movimiento del jugador o invocar el menú. En el primero de los casos cuando el usuario deja de presionar una tecla de movimiento, el incremento de X o Y pasa a valer 0 y por tanto el personaje deja de moverse. Si el usuario deja de presionar la tecla de menú, entraremos en modo menú de historia.

4.2.2. Sistema de Menús

Se han desarrollado tres menús diferentes para la gestión del juego, cada uno de ellos corresponde a una instancia de la clase Menu (ver anexo 6). Los tres menús corresponden a diferentes modos de ejecución en los que se puede encontrar la aplicación, Inicio, Historia y Batalla.

Menú de inicio:

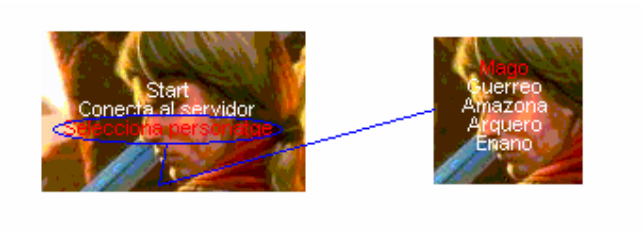

Figura 4.5. Menú de inicio.

En el menú de inicio, se elige el personaje que representará al jugador local durante la partida, posteriormente se conecta al servidor, que informará a los demás jugadores de la posición y habitación que ocupa el nuevo jugador. Posteriormente se empieza la partida y se pasa al modo historia.

Menú de historia:

Figura 4.6. Menú del modo historia.

A través del menú de historia se puede realizar la petición, al servidor, tanto de unión como de ataque a otro jugador, una vez elegida la acción a realizar, se invoca al selector para determinar que rival será el objetivo de la acción.

Menú de batalla:

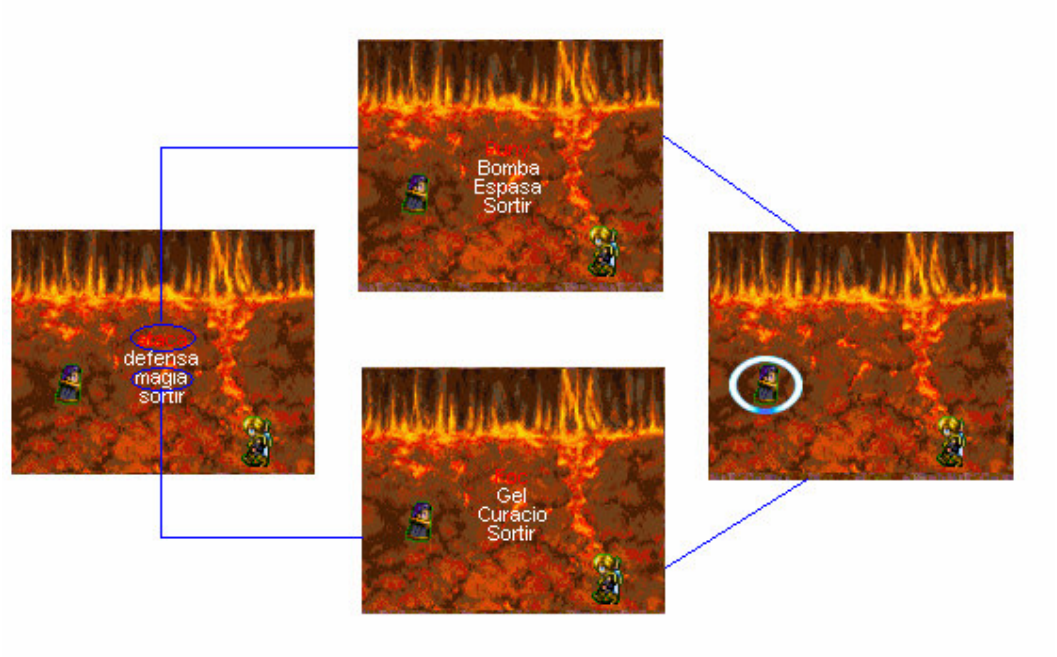

Figura 4.7. Menú del modo batalla.

En el menú de batalla se elegirá el arma a utilizar en el ataque, la técnica defensiva a realizar o el hechizo a formular una vez realizada la elección se invoca al selector para determinar el objetivo.

Este menú se habilita siempre que dispongamos del turno de ataque.

4.2.3. La clase Menu

Es la clase empleada para crear menús, añadirles opciones y registrar cual de esas opciones es la resaltada en cada momento, pudiendo ser actualizada desde el método keyReleased() de la clase SSCanvas.

El método keyReleased() es el encargado de escuchar todos los eventos de teclado, por tanto si alguno de los tres menús se encuentra activo, y el usuario libera las teclas de navegación, se invoca al método correspondiente para "subir" o "bajar" el selector por la lista de opciones, estos métodos de la clase Menu son Puja() y Baixa().

Cuando un menú se encuentra activo, en la clase principal SSCanvas se activa un "flag", mediante el cual en el método keyReleased() se puede identificar el objeto Menú con el que se va a interactuar.

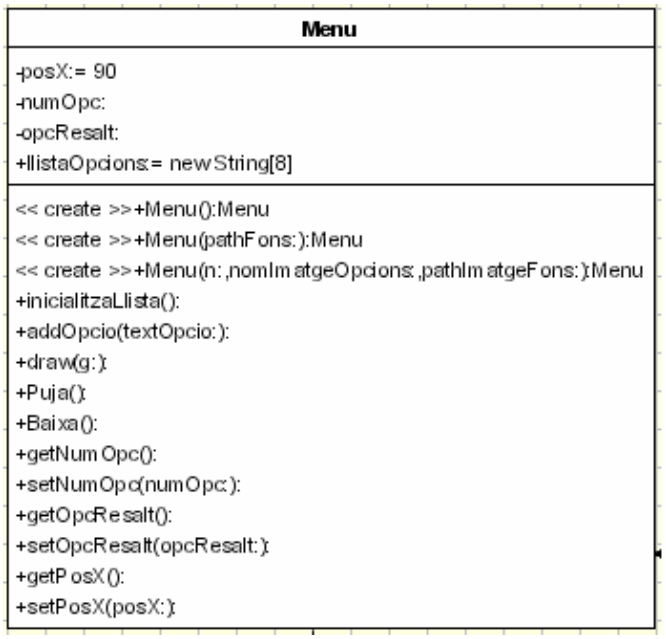

Figura 4.8. Diagrama UML de la clase Menu.

Por ejemplo, si estando en modo de historia el jugador pulsa la tecla "3", activaría el flag del menú de historia, el método paint() de SSCanvas detectaría

que el menú está activado y representaría en pantalla las diferentes opciones del menú, resaltando una de ellas como opción escogida. Utilizando las teclas de dirección del dispositivo, más concretamente la tecla “Up” y la “Down”, cambiaría la opción resaltada. Finalmente si el usuario pulsa la tecla de acción se elige la opción resaltada, ejecutando el código que tenga asociado esa opción. En el caso de escoger, por ejemplo, la unión a otro jugador, se activaría el selector par indicar cual seria el jugador objetivo de la unión, ese seria el código asociado a la opción.

4.2.4. Animación gráfica

Cuando el listener keyPresed() detecta que el usuario a pulsado una tecla de movimiento, incrementa la posición en la dirección y sentido de la tecla pulsada y en el próximo refresco de la imagen de pantalla el personaje aparecerá representado en la nueva posición, en cada aumento de posición que se realiza en la misma dirección y sentido, el frame a dibujar en pantalla cambia dando la sensación de que el personaje anda. Vemos un ejemplo a continuación.

Figura 4.9 Secuencia de animación.

En el modo de batalla, se utiliza esta misma técnica de animación para crear la sensación de que el personaje anda, aunque en este caso el usuario no tiene un control directo sobre el movimiento del personaje, puesto que el tan solo determinará un objetivo, una posición hasta la que el personaje debe llegar, y de la que también debe regresar. Una vez el usuario ha escogido un ataque y un objetivo, el gestor de batalla, es el encargado de determinar el camino que debe seguir el personaje hasta su objetivo, una vez alcanzada la posición del objetivo, el gestor de batalla ejecutará la animación correspondiente al tipo de arma utilizado para el ataque. Una vez finalice la animación del ataque, se procederá a restar los puntos de vida y magia correspondientes en los indicadores de juego y posteriormente el jugador regresará a su posición de batalla original.

El gestor de batalla es también el encargado de recrear los ataques que realizan los demás jugadores, para ello el módulo de comunicación facilitará el identificador del jugador que realiza el ataque, el jugador que es objeto de dicho ataque y el tipo de arma empleado para el ataque. Una vez el gestor de batalla ha obtenido esta información procede a actualizar la situación de batalla y se activa la animación del ataque del jugador atacante hacia el jugador objetivo.

Para recrear los ataques, se ha utilizado un objeto Sprite que almacena los diferentes frames que representan cada instante del ataque en la animación. Mediante la repetición cíclica de todos los frames de manera ordenada, conseguimos un efecto visual que representa de manera simbólica un ataque. El la siguiente secuencia, se muestran todos los frames de un ataque.

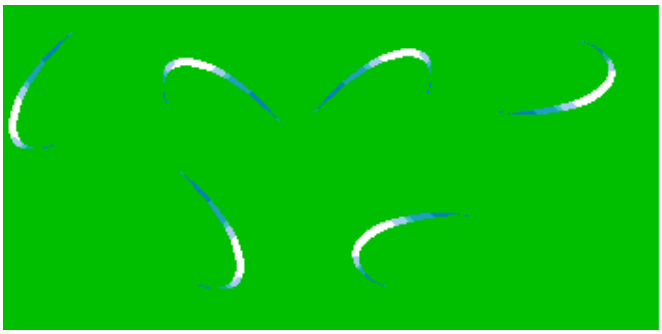

Figura 4.10. Secuencia animación de ataque.

Mediante un contador que se incrementa cada vez que se muestra un frame del ataque por pantalla, se controla el frame a seleccionar del Sprite en cada momento, cuando este contador alcanza una cifra igual al número de frames de la animación, se resetea y vuelve a mostrar la misma secuencia, la duración de la animación se controla con un contador que se decrementa con cada secuencia mostrada por pantalla.

De igual manera que el gestor de batalla ha de controlar las animaciones de los jugadores rivales en el modo de batalla, existe un gestor de jugadores rivales en el modo de historia encargado de actualizar las posiciones de los rivales según la información que recibe del modulo de comunicación. Este gestor se encarga también de seleccionar el frame del jugador rival a mostrar en cada momento.

4.2.5. Hilo principal del juego

La clase principal SSCanvas implementa la interfaz Runnable, esto implica que debe implementar un método run() (ver anexo 7), que será invocado de forma cíclica por la ejecución del programa, es por tanto dentro de éste método donde se deben implementar todas aquellas llamadas a métodos que sean necesarias para la ejecución de la aplicación. Para una correcta gestión del juego, el hilo principal ejecuta de forma periódica tareas tales como actualizar las posiciones y frames del jugador local en función de los incrementos de X e Y, actualizar los datos que recibe el módulo de comunicación y procesarlos, enviar los datos de actualización al servidor y pintar por pantalla la situación actual de juego. En cada ciclo de ejecución del hilo principal, se realiza una llamada al método

sleep() de la interfaz para proporcionar un margen de relajación en la ejecución de la aplicación.

En función del modo actual en que se encuentra el juego, el hilo principal ejecuta una parte de código distinta y una común a los tres modos de juego. Si el juego se encuentra en modo historia, se actualiza al jugador local según los eventos de teclado recibidos, se selecciona el mapa a mostrar en función de la habitación en la que se encuentra el jugador, se envían los datos de actualización al servidor y finalmente se evalúan los datos recibidos del modulo de comunicación y se actualizan los jugadores rivales.

Si por el contrario nos encontramos en modo de batalla, en primer lugar se evalúan los datos del modulo de comunicación, se llama al método que se encarga de actualizar el estado de la batalla dentro del gestor de batalla.
En el caso de encontrarnos en modo inicio, se escucha el modulo de comunicación para recibir el paquete inicial del servidor que nos permite actualizar la posición X e Y inicial del jugador local, así como la habitación en la que se encuentra al principio del juego.

4.2.6. Frecuencia de refresco

La frecuencia de refresco de una animación renderizada se mide en fps (frames por segundo) este valor corresponde a la cadencia con la que una nueva imagen es representada en pantalla. Cada vez que se llama al método repaint() durante la ejecución del juego, se ejecuta el método paint() que dibuja por pantalla el frame que debe mostrarse en pantalla en cada momento del juego. Se necesita un parámetro que permita ajustar la frecuencia de refresco de imagen en función de la velocidad de procesado del dispositivo en el cual se ejecute la aplicación. Cada dispositivo ofrece una velocidad de procesado diferente por lo que deberemos ajustar el valor del parámetro que permite controlar la velocidad de refresco para cada dispositivo en el que se desee ejecutar el juego. En nuestro caso el parámetro que controla la frecuencia de refresco es el retardo en la ejecución del hilo principal que se introduce con la llamada al método Sleep() en cada ciclo de ejecución.
Para que el ojo humano tenga la sensación de movimiento al observar una secuencia de imágenes superpuestas, las tasa de refresco de imagen debe ser superior a los 16 fps.
El ajuste óptimo del número de frames por segundo a mostrar por pantalla en función del retardo introducido por el método Sleep(), se ha realizado de forma totalmente empírica dando diferentes valores al retardo y realizando pruebas del juego. El valor del parámetro que se pasa a Sleep() para introducir un retardo en esta aplicación es de 180. Con este valor, el juego se ejecuta correctamente en el simulador del entorno de desarrollo.

4.2.7. Método paint()

Como se ha comentado con anterioridad, el método paint() es el encargado de mostrar por pantalla la situación de juego en cada instante. Dentro de dicho

método se diferencia entre el modo de ejecución en el que nos encontramos, para ejecutar una parte de código u otro, puesto que cada modo de ejecución necesita prestaciones distintas.

En el modo de inicio, se muestra por pantalla el fondo de presentación del juego con el menú de inicio superpuesto. Una vez seleccionada la opción "Start" en el menú de inicio, se pasa al modo historia, donde el método paint() se encarga de dibujar el mapa en el que se encuentra el jugador, el marco de indicadores o interfaz de usuario y finalmente los jugadores tanto local como rivales. Si se invoca el menú en el modo historia, se para la animación de los jugadores y el programa atiende solamente al menú. Si nos encontramos en el modo batalla, la composición de pantalla es la misma que en el modo historia salvo en las animaciones que representan los ataques, estas animaciones son superpuestas a la imagen del personaje que es objeto del ataque. El modo de batalla también dispone de un menú que el método paint() deberá encargarse de dibujar en pantalla cuando éste esté activo.

4.3. Gestión del Juego

La clase SSCanvas es la encargada de centralizar y controlar los diferentes gestores de cada aspecto del juego. SSCanvas crea las instancias de todos los gestores y durante el tiempo de ejecución intercambia datos con todos ellos. SSCanvas gestiona además todas las interacciones con el usuario, es la clase encargada de escuchar eventos del teclado y de dibujar imágenes en la pantalla.

Tanto el gestor de batalla como el gestor de jugadores rivales en modo historia emplean instancias de la clase JugadorRivals para almacenar y actualizar los parámetros y variables de cada uno de los jugadores. Cada jugador está representado por un objeto de la clase Rival, esta clase contiene el Sprite que representa gráficamente al jugador, la posición que ocupa en el mapa y la habitación en la que se encuentra.

En el inicio del juego, el servidor asigna un identificador a cada jugador que se conecta a la partida. Éste es un hecho fundamental para la gestión del juego puesto que se utiliza esta id para identificar al mismo jugador en cada cliente del juego, así como en el servidor. Cuando el jugador local recibe un paquete de acción de cualquier jugador, lo actualiza en el gestor de jugadores rivales. Para identificar a que jugador pertenece cada paquete, se envía el número de id en el mismo, con lo que el gestor de rivales únicamente debe recorrer el vector donde se almacena la información referente a los rivales y comparar el identificador de cada jugador con el del paquete recibido, si coincide actualiza los datos de ese rival. La clase Batalla procede del mismo modo para gestionar a todos los jugadores que toman parte en cada enfrentamiento.

En la siguiente figura se observa el diagrama de clases que muestra todas las clases que intervienen en la gestión del juego. Puede verse con claridad como SSCanvas es la clase principal en cuanto a gestión de juego, en ella se

instancian los objetos del resto de clases que intervienen en el desarrollo del juego.

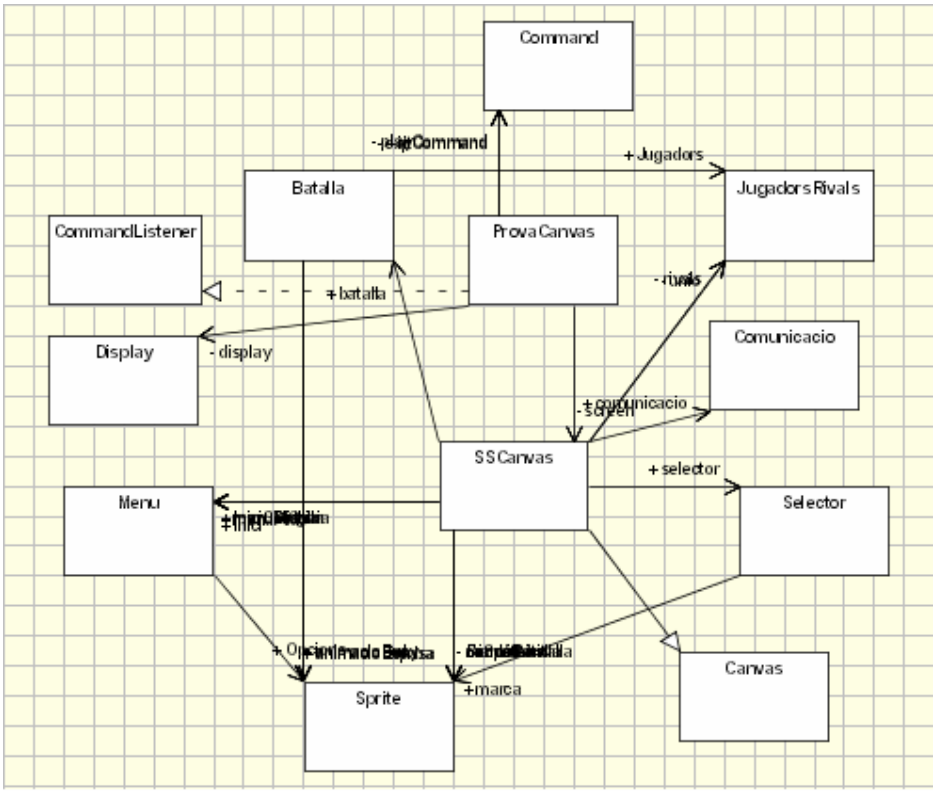

Figura 4.11. Diagrama de clases del cliente.

4.3.1. Clase JugadorsRivals

Esta clase permite almacenar información de un número concreto de jugadores, mediante un vector de objetos Rival, se almacena y actualiza la información referente a todos los jugadores gestionados por esta clase JugadorsRivals.

Tanto la clase Batalla como la clase SSCanvas crean una instancia de esta clase, en la clase Batalla se utiliza para almacenar y gestionar información de todos los jugadores que toman parte en una batalla. En el caso de SSCanvas se crea una instancia de JugadorsRivals que permite la gestión de todos los jugadores rivales en el modo historia.

JugadorsRivals ofrece métodos que permiten agregar rivales al vector, eliminarlos del mismo, inicializar los parámetros de cada jugador, cargar los frames correspondientes a cada personaje en el objeto rival y actualizar la situación de cada jugador.

4.3.2. Clase Batalla

Cuando el juego se encuentra en modo de historia y se recibe del módulo de comunicación un paquete de orden de batalla, desde SSCanvas se instancia un objeto Batalla, se le añaden todos los jugadores que van a tomar parte en la batalla y se arranca el thread o hilo de ejecución de dicha clase.

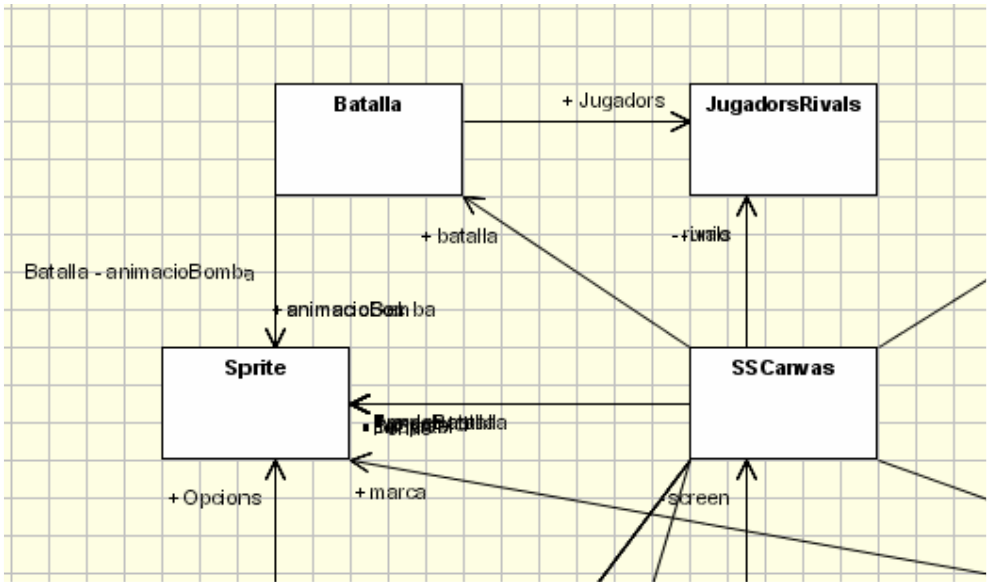

Figura 4.12. Diagrama de clases relacionadas con Batalla.

En el diagrama de clases de la figura anterior, podemos observar la relación entre Batalla, SSCanvas, JugadorsRivals y Sprite, las diferentes clases que toman parte en la gestión de batalla. SSCanvas instancia un objeto de la clase Batalla, se crea el gestor de batalla cuando se recibe la orden desde el módulo de comunicación. La clase Batalla, instancia un objeto de la clase JugadorsRivals mediante el cual gestionará y actualizará la información relativa a los participantes en la batalla, finalmente Batalla crea diferentes instancias de la clase Sprite donde se almacenan las diferentes animaciones específicas de cada ataque.

La Clase Batalla implementa la interfaz Runnable tal como hacia la clase SSCanvas, por tanto también debe implementarse el método run(), que se repetirá cíclicamente mientras el objeto Batalla se encuentre activo.

Batalla posee un vector para almacenar los objetos rival que representan a cada jugador implicado en la batalla, mediante el método ActualitzaBatalla(), se mantiene y actualiza la información necesaria de cada jugador, la clase SSCanvas realiza una llamada a este método cada vez que se recibe un paquete de orden de ataque desde el módulo de comunicación y se le pasa por parámetro el contenido de dicho paquete, es por tanto Batalla la clase

encargada de gestionar la posición de cada jugador en cada momento, en función de la información recibida de la clase principal.

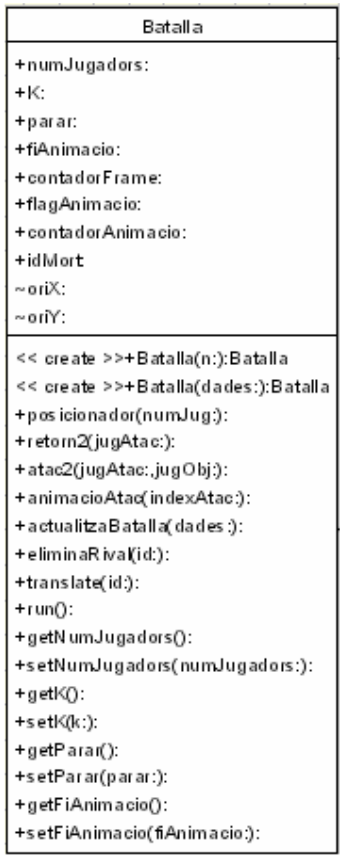

Figura 4.13. Diagrama UML de la clase Batalla.

Batalla implementa una lógica que permite gestionar el movimiento de cada jugador en función de si éste ha lanzado un ataque. Cuando se recibe una orden de ataque, se activa un mecanismo que lanzará y gestionará la animación que representa dicho ataque, para ello dispone de cuatro métodos que conforman la lógica de batalla. Estos cuatro métodos son : atac(), animacioAtac(), retorn() y posicionador(). Al recibir el paquete de orden de ataque en SSCanvas, se activa el flag de ataque en el objeto rival correspondiente al jugador que lanza el ataque, del mismo modo se marca el atributo objetivo de dicho objeto con el valor del identificador del jugador objetivo. Durante la ejecución del método run() de la clase batalla, se recorre el vector de jugadores en busca de un jugador cuyo flag de ataque se encuentre activado una vez localizado, se invoca el método atac(), pasando por parámetro la identificación del jugador atacante y del jugador objetivo y se inicia la animación, en cada ciclo de ejecución, se modifica la posición del jugador atacante para que éste alcance la posición del jugador objetivo, una vez alcanzada esta posición se realiza la llamada al método animacioAtac(), en el próximo ciclo de ejecución se inicia la animación del ataque, el jugador

mantiene su posición y se representa en pantalla la sucesión de imágenes que conforman el ataque, una vez ha finalizado la animación de dicho ataque, se realiza la llamada al método retorn(), encargado de que el jugador atacante vuelva a ocupar su posición original en el escenario de batalla, una vez el jugador ha regresado a su posición, se invoca el método posicionador() para colocar de forma definitiva al jugador atacante en la posición y orientación inicial en el escenario y éste pasa a esperar los ataques del resto de jugadores.

4.3.3. Unión de jugadores

Para la gestión de la unión de jugadores, la clase SSCanvas, instancia un objeto de la clase JugadorsRivals mediante la cual en la que se almacena un vector de jugadores, formado por el jugador local y el jugador al que éste se ha unido. Para la unión, el jugador local debe enviar un paquete de petición de unión al servidor, indicando la id del jugador al que se va a unir y la suya propia cuando el servidor recibe el paquete de unión, informa tanto al solicitante como al jugador al que se va ha unir de que va a producirse la unión, en ese momento el listener del jugador local, encargado de atender los eventos de teclado, únicamente escuchará la tecla de desunión y la de invocación del menú. Por tanto de lo único que debe preocuparse el gestor de jugador local, es de actualizar su posición a la misma que el jugador al que se ha unido más un cierto desplazamiento y de informar de dicha posición al servidor.

Cuando el jugador local unido a otro jugador, decide desvincularse de él, envía otro paquete al servidor, solicitando la desunión y de la misma forma que en el caso anterior el servidor informa a los dos jugadores. Antes de que se realice la desunión, se actualiza la posición del gestor de jugador local con los parámetros del gestor auxiliar creado para la unión si se sigue el desarrollo normal del juego, atendiendo los eventos de movimiento generados desde el teclado.

La solicitud de unión a otro jugador, se realiza desde el menú de historia, una vez invocado este menú, se selecciona la opción “unió” del menú y seguidamente aparece el selector para elegir a que personaje va a unirse el jugador local. Es una vez realizada la selección cuando se envía al servidor el correspondiente paquete con la id del jugador local y la del jugador con el que se va a realizar la unión.

4.4. Comunicaciones

4.4.1. Clase Server y multithreading

Para el desarrollo de cualquier aplicación multiusuario que centralice la información aportada por todos los usuarios, la procese, genere nueva información en función de la recibida y posteriormente la distribuya a diferentes usuarios, necesita una entidad que central a la que todos los elementos de la comunicación estén conectados. En este caso se ha desarrollado un servidor

que realiza todas estas funciones y que deberá ser ejecutado en un computador con conexión a internet, este servidor debe atender a diferentes entradas y salidas de información de forma prácticamente simultánea, es ésta la razón por la que se ha optado por la implementación de una aplicación multihilo o multitheading que ejerza las funciones de servidor de juego.

El funcionamiento del servidor en este proyecto, es sencillo, permanece a la escucha del puerto designado para la comunicación, cuando recibe un paquete, evalúa el tipo de paquete recibido, procesa dicho paquete en función del tipo y genera un paquete que será enviado a uno o varios de los usuarios conectados al servidor en función de la información contenida por dicho paquete.

En el siguiente esquema se puede ver el diagrama de clases del servidor:

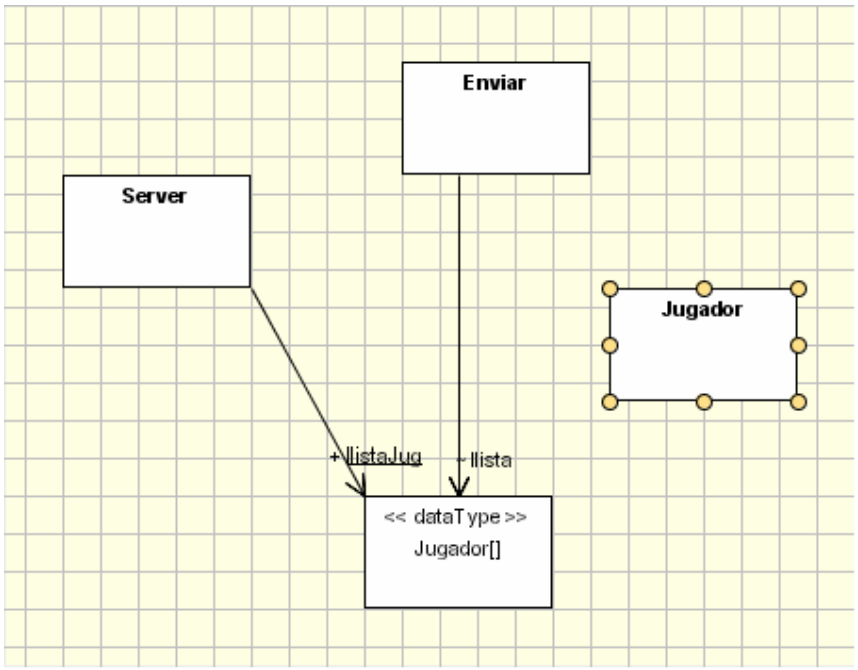

Figura 4.14. Diagrama de clases del servidor.

Para la correcta gestión del juego, el servidor debe tener información del estado de todos los jugadores conectados a la partida en cada momento, para ello se ha implementado la clase Jugador. La clase Jugador almacena información relativa a un jugador de la partida, dicha información hace referencia a la posición, la habitación que ocupa, los puntos de vida y de magia…

A continuación se muestra el esquema de métodos y atributos de la clase Jugador.

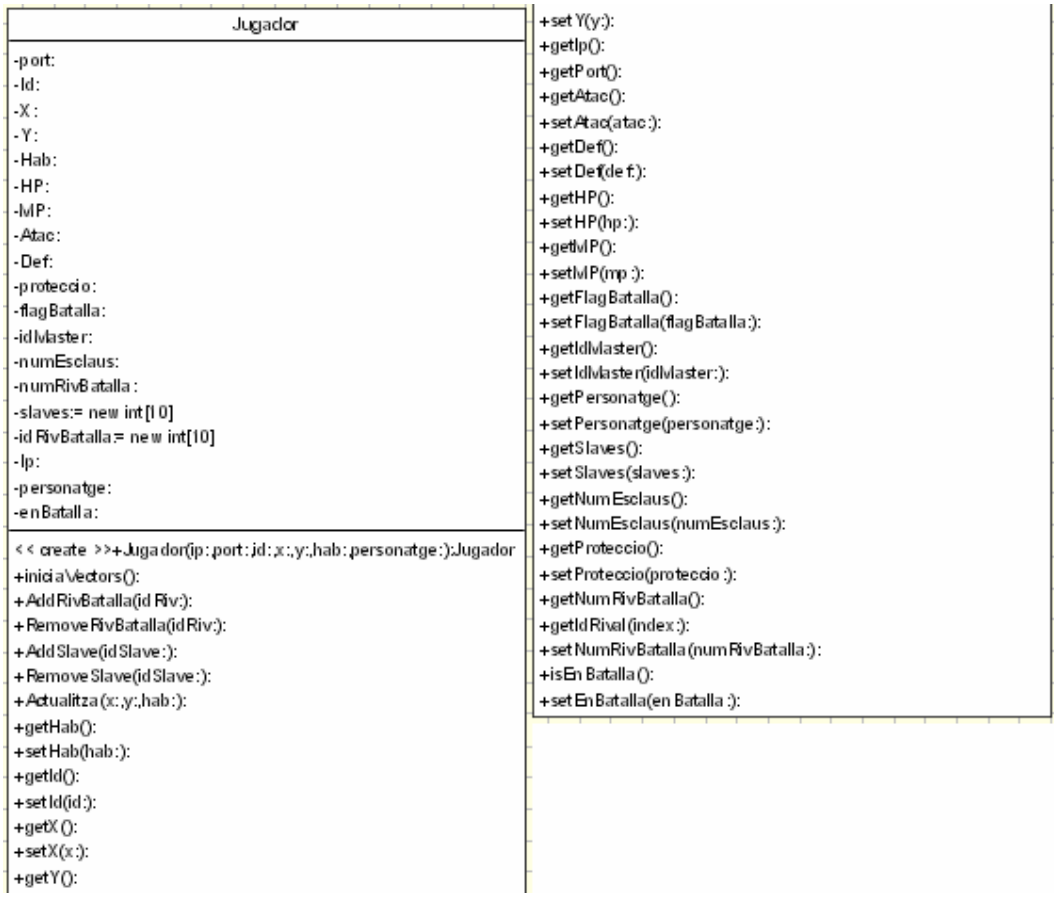

Figura 4.15. Diagrama UML de la clase Jugador.

La clase Server, tiene un vector de objetos de la clase Jugador, cuando el servidor recibe un paquete de tipo “Alta” de un nuevo jugador, crea una instancia de la clase Jugador y lo añade al vector después de inicializar sus atributos, asignándole un identificador y una posición y habitación de inicio de juego, una vez dado de alta el jugador, el servidor envía a dicho jugador el paquete de inicialización para que éste sepa en que posición y habitación del mapa debe empezar.

El servidor es el encargado también de gestionar todas las batallas que tengan lugar a lo largo del juego, cuando recibe un paquete de “Ataque”, el servidor evalúa la información actual de los dos jugadores que toman parte en el ataque, es decir el jugador atacante y el jugador objetivo. Al tratarse de un juego de rol, el resultado de cada ataque se decide en función de las características propias del jugador atacante y del jugador objetivo, a la evaluación de dichas características se añade un factor de “suerte”, generado a través de una variable aleatoria con el fin de que cada ataque tenga un

resultado diferente y no pueda ser conocido con exactitud por los jugadores antes de realizar el ataque.

Una vez obtenido el resultado del ataque, el servidor actualiza la información que hace referencia a los puntos de vida del jugador objetivo y a los puntos de magia del jugador atacante. Cuando la información de los jugadores ha sido actualizada en el servidor, éste envía a todos los jugadores que toman parte en la batalla un paquete de actualización que contiene la información necesaria para que cada uno de ellos actualice sus puntos de vida y magia y pueda representar por pantalla la animación del ataque.

En el modo historia, el servidor realiza únicamente funciones de repetidor o concentrador a parte de actualizar la posición de todos y cada uno de los jugadores. Cuando el servidor recibe un paquete de “Acción” del modo historia, actualiza la posición X e Y del jugador que envía el paquete, así como la habitación en la que se encuentra. Cuando ha actualizado los datos del jugador, envía al resto de jugadores de la partida un paquete de actualización con la posición y la habitación de dicho jugador que ha realizado el movimiento.

4.4.2. La Clase Enviar

El servidor permanece de manera continua a la escucha del puerto de comunicaciones, por lo que necesita de una clase auxiliar para generar el tráfico de salida. La clase enviar, es la encargada de generar los paquetes de comunicación y enviarlos al destino que le indique el servidor. La clase Enviar implementa la interfaz Runnable por lo que cada vez que se empieza la ejecución de un objeto de dicha clase, se invoca al método run. Cada vez que el servidor necesita enviar un paquete a un cliente, instancia un objeto de la clase Enviar como un nuevo hilo de ejecución mediante un objeto Thread, cada uno de estos hilos o hebras se ejecutan de forma independiente del resto de hilos de ejecución del servidor, por lo que podemos atender a diferentes clientes a la vez.

La clase Enviar dispone de siete constructores diferentes que se invocan en función del tipo de paquete que se debe enviar al cliente, este hecho nos permite gestionar todo el tráfico saliente con una única clase que actuará de forma distinta en función del método constructor invocado en cada caso.

Los diferentes métodos constructores de la clase Enviar se distinguen por los argumentos que se pasan por parámetro al constructor desde la clase Server debido a que en cada caso necesita diferente información para generar el paquete que debe enviar al cliente.

El hecho de que una clase disponga de métodos con el mismo nombre, pero que reciben diferentes argumentos y ejecutan código diferente en función de dichos argumentos, se denomina sobrecarga de métodos y permite que una misma clase actúe de formas diferentes al invocar al mismo método en función de los argumentos que se le pasen.

A continuación se muestra el esquema que representa a la clase Enviar:

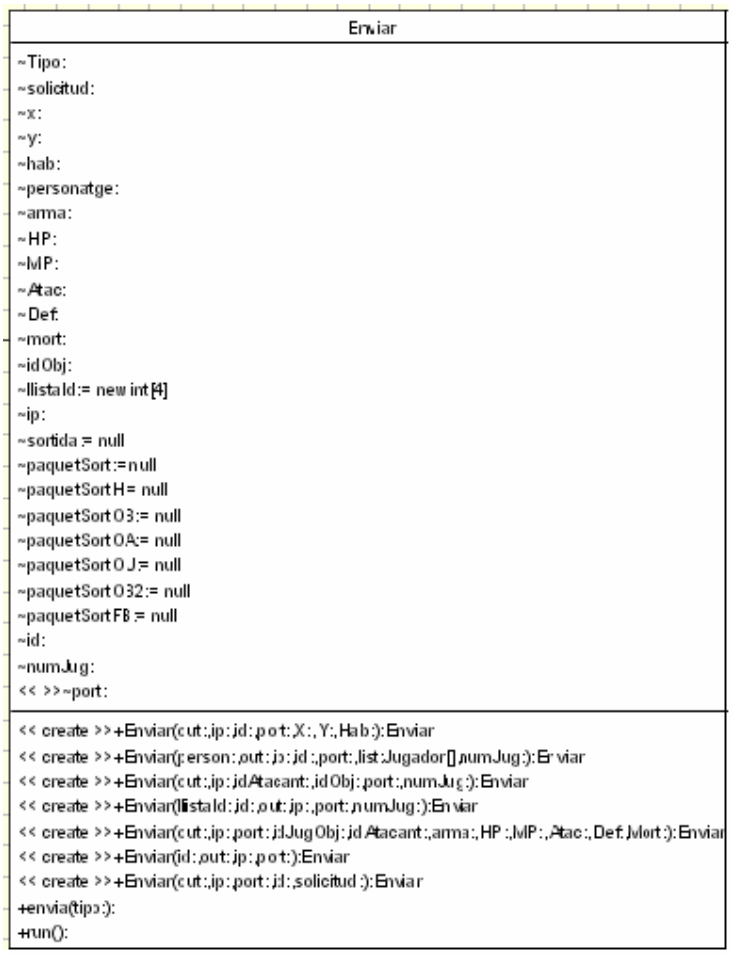

Figura 4.16. Diagrama UML de la clase Enviar.

En el diagrama se pueden observar los distintos constructores de los que dispone la clase Enviar y como únicamente existen dos métodos más, el método run() que se invoca al iniciar la ejecución del objeto, y el método envia() (ver anexo 8), que es el encargado de generar el tipo de paquete requerido por la clase Server y enviarlo al cliente correspondiente, este método se invoca desde el método run().

4.4.3. Clase Comunicacio

El cliente, necesita un módulo de comunicación para poder conectarse e interactuar con el resto de clientes a través del servidor. A diferencia del servidor, el módulo de comunicaciones del cliente no necesita atender diversas peticiones del resto de clientes, puesto que todo el tráfico está centralizado por el servidor, es por ello que no es necesaria la implementación de una solución multithreading en la clase Comunicacio, sin embargo si es necesaria la

implementación de la interfaz Runnable, puesto que la aplicación necesita ejecutar diferentes módulos de forma simultánea. La clase Comunicació permanece a la escucha del puerto de conexión al servidor actualizando, con la información recibida del servidor, un vector de valores numéricos (int), a través del cual se obtienen los datos que solicita la clase principal SSCanvas.

Los métodos recibirDatagrama() y leerDatagrama(), son los dos métodos encargados de recibir e interpretar el flujo de paquetes entrantes, el primero de los métodos, transforma el tráfico recibido en un array de bytes que se pasará al segundo método, éste interpreta el vector de bytes y lo transforma en un vector de enteros al que accederá la clase principal de la aplicación cuando necesite actualizar los datos de comunicación dentro del bucle principal del juego.

La clase se encarga además de la recepción de información que proviene del servidor, al envío de paquetes al mismo, para ello se utilizan diferentes métodos en función del tipo de paquetes, para la creación de dichos paquetes se utiliza un método inverso al caso de la recepción, se crea un String con la información a enviar, se convierte a un flujo de bytes y se envía con un datagrama UDP. Los métodos encargados del envío de paquetes se pueden ver en el siguiente esquema de la clase Comunicacio.

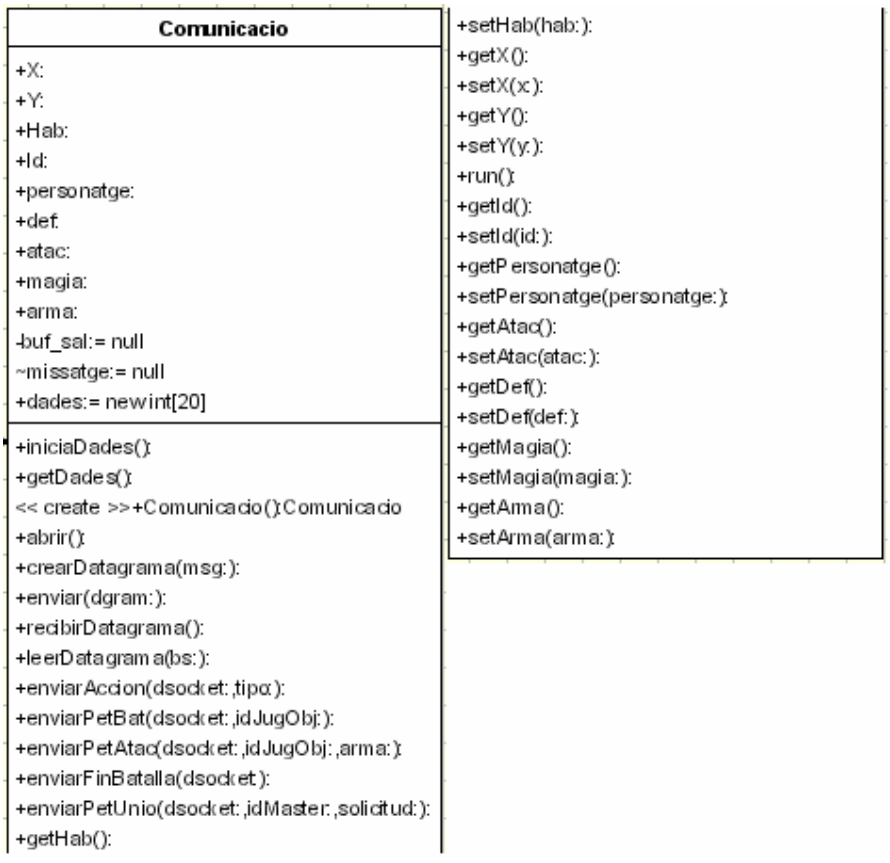

Figura 4.17. Diagrama UML de la clase Comunicacio.

CAPITULO 5. POSIBLES AMPLIACIONES

5.1. Sincronismo

Para adaptar el juego a un entorno real de comunicaciones, uno de los aspectos fundamentales a tener en cuenta para su correcto funcionamiento es el sincronismo, se debería desarrollar un módulo tanto en el cliente como en el servidor que permita que todos los jugadores “vean” en todo momento el mismo instante de juego. Para ello debería tenerse en cuenta la posibilidad de que los paquetes lleguen desordenados o con cierto retardo que debe ser corregido.

5.2. Tolerancia a pérdidas y persistencia

Otro factor que debe tenerse en cuenta en un entorno real de ejecución es la pérdida de paquetes. Una aplicación en tiempo real no debe tener que esperar al reenvío de un paquete para continuar la ejecución de la aplicación correctamente puesto que se perdería la linealidad de la historia o del juego. El juego ha de poder asumir cierto porcentaje de pérdida de paquetes en la comunicación por la red sin que ello afecte a su correcto funcionamiento. Para la adaptación del juego a un entorno real es fundamental diseñar un protocolo de comunicación que tenga en cuenta dicho aspecto e implemente por ejemplo cierta redundancia en la información contenida en diferentes paquetes, para que no perder la persistencia de juego con la pérdida de paquetes.

5.3. Búsqueda de servidores

Éste es uno de los campos que ha tenido un mayor desarrollo en el mundo del videojuego en los últimos tiempos. El hecho de que la mayoría de juegos permitan un modo multijugador ha provocado la aparición de aplicaciones de gestión de partidas online y de búsqueda de servidores de juego como puedan ser Steam, GameSpy o The all-seeing game.

En la aplicación que se ha desarrollado se indica en el código, la ip del servidor al que debe conectarse el programa cliente para unirse a una partida. Una posible e interesante ampliación del proyecto seria el desarrollo de una plataforma que permita la búsqueda de distintas partidas a las que conectarse a lo largo de la red.

5.4. GPRS vs. Bluetooth

Otra posible ampliación de este proyecto consiste en adaptar el protocolo de comunicaciones para permitir la conexión a través de bluetooth entre diferentes jugadores utilizando un ordenador como servidor o adaptando la aplicación para eliminar la necesidad de un servidor que centralice el tráfico. Bluetooth ofrece varias ventajas frente a GPRS como por ejemplo el coste de conexión, puesto que BT crea una red privada, no es necesaria la conexión a través de un ISP (Internet Service Provider). Otra ventaja que ofrece BT es que reduce de forma considerable el retardo en la transmisión de paquetes respecto a GPRS y por tanto reduce la latencia de juego. La principal limitación introducida por una solución basada en Bluetooth hace referencia al número de usuarios que se pueden interconectar. Un dispositivo BT puede conectarse con siete dispositivos más para formar una pico-net como máximo.

Referencias bibliográficas.

Sitios Web:

http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/introduccion/indice2.html

http://java.sun.com

http://www.todosymbian.com

http://www.gsmspain.com/j2me/

http://cs1.comunidad-siemens.com/index.php

http://charas-project.net/

http://www.mailxmail.com/curso/informatica/java

http://www.developer.com/java/j2me/

http://eclipseme.org/index.html

http://www.programacion.com/tutorial/ags_j2me/

http://www.sourceforge.net

http://www.sourceforge.net

Libros:

[1] Sergio Gálvez Rojas y Lucas Ortega Díaz. *Java a tope: java 2 micro edition.* "Edición electrónica."

[2] Gabriel Alberto Vásquez Muñoz y Euler Adrián Trejo Narváez
Plataforma Integrada de Desarrollo de Aplicaciones para Dispositivos Móviles con J2ME. "Trabajo de Grado"

Anexos:

En la mayoría de los códigos fuente que se muestran a continuación se han omitido los métodos Getters y Setters de los atributos.

Anexo 1: Código de la clase Comunicacio.

```
public class Comunicacio implements Runnable{

     public int X,Y,Hab,Id,personatge;
     public int def,atac,magia, arma;

      private byte[] buf_sal = null;


       DatagramConnection sender = null;
       DatagramConnection receiver = null;
```

```
        Datagram datagrama = null;
        String missatge = null;
        public int[] dades = new int[20];



     public void iniciaDades(){
          for(int i=0;i<10;i++){
               dades[i]=-1;
          }
     }
     public int[] getDades() {
          return dades;
     }
     public Comunicacio()throws IOException {

          missatge = "Comunicación ok \n";
          //personatge = 0;
         abrir();

          Id=-1;
          arma =0;
     }

public void abrir() throws ConnectionNotFoundException{

//abro la conexión sender con 127.0.0.1:5555

          try {
          sender = (DatagramConnection)
          Connector.open("datagram://127.0.0.1:5555");

           }
           catch (IOException ex) {
           }

 }


public void recibirDatagrama() throws IOException {

 byte[] buffer = new byte[256];
Datagram dgram = sender.newDatagram(buffer, buffer.length);

int t =0;
while (t==0) {
      dgram.setLength(buffer.length);
      sender.receive(dgram);
      int length = dgram.getLength( );
      String msg = new String (dgram.getData());
      leerDatagrama(dgram.getData());
```

```
      System.out.println("datagrama rebut: " + msg);
      t=1;
      }
 }

 public void leerDatagrama(byte[] bs) throws IOException {
          String info = null;
          String cadena = new String(bs);
          int contador;
          contador=0;
          int i;
          i=0;
          int t=0;
          char[] prova = new char[20];
          prova = cadena.toCharArray();
          while(t!=-1){


         if((prova[i] == '/')&&(cadena.length()>0)){
          info=cadena.substring(0,i);
          cadena = cadena.substring(i+1,cadena.length());
          dades[contador] = Integer.parseInt(info);
          contador++;
          if(prova[i+1]=='/')t=-1;
             prova = cadena.toCharArray();
             i=-1;
          }
         i++;
         }

         if(dades[0]==1){
          Id = dades[1];
         }
}

public void enviarAccion(DatagramConnection dsocket, int
tipo) throws IOException {

     try{
     String datosJuego= new String
     (tipo+"/"+X+"/"+Y+"/"+Hab+"/"+Id+"/"+personatge+"/"+de
     f+"/"+atac+"/"+magia+"/"+"/");
     byte[] msn_juego = datosJuego.getBytes();
     Datagram dp =
     sender.newDatagram(msn_juego,msn_juego.length);
                    dsocket.send(dp);
     }
     catch (Exception e){
     System.err.println("Error enviando paquete");
     }
}
```

```
public void enviarPetBat(DatagramConnection dsocket, int
idJugObj)throws IOException {

     try{
     String datosPeticion = new String
     ("3"+"/"+"0"+"/"+idJugObj+"/"+Id+"/"+"/");

     byte[] msn_juego = datosPeticion.getBytes();
     Datagram dp = sender.newDatagram(msn_juego,
     msn_juego.length);
     dsocket.send(dp);
     }
     catch (Exception e ){
     System.err.println("imposible enviar peticion
     batalla");
     }
}

public void enviarPetAtac(DatagramConnection dsocket, int
idJugObj, int arma)throws IOException {
     try{
     String datosAtac = new String
     ("3"+"/"+"1"+"/"+idJugObj+"/"+Id+"/"+arma+"/"+"/");
     byte[] msn_juego = datosAtac.getBytes();
     Datagram dp =
     sender.newDatagram(msn_juego,msn_juego.length);
     dsocket.send(dp);
     }catch (Exception e){
     System.err.println("imposible enviar peticion
     ataque");
     }
}

public void enviarFinBatalla(DatagramConnection
dsocket)throws IOException {
     try{
     String datosFinBatalla = new String
     ("3"+"/"+"9"+"/"+Id+"/"+"/");
     byte[] msn_juego = datosFinBatalla.getBytes();
     Datagram dp =
     sender.newDatagram(msn_juego,msn_juego.length);
     dsocket.send(dp);

     }catch (Exception e){
     System.err.println("error al enviar paquete
     finBatalla");
```

```
     }
}



public void enviarPetUnio(DatagramConnection dsocket, int
idMaster, int solicitud){
     try{
     String datosUnio = new String
     ("4"+"/"+solicitud+"/"+Id+"/"+idMaster+"/"+"/");
     byte[] msn_juego = datosUnio.getBytes();
     Datagram dp = sender.newDatagram(msn_juego,
     msn_juego.length);
     dsocket.send (dp);
     }
     catch  (Exception e){
     System.err.println("imposible enviar peticion unio");
     }
}

public void run(){
     while(true){
          try {
               recibirDatagrama();
               } catch (IOException e) {
                    // TODO Auto-generated catch block
               e.printStackTrace();
               }
     }
}
```

Anexo 2: Código de la clase Sprite comentado.

La clase Sprite es fundamental para la carga de la imágenes que representan a los personajes, a las animaciones de ataques y los fondos de pantalla e indicadores.

Como se puede observar, el constructor necesita que se le pase como argumento el número de frames que formaran el Sprite para dimensionar correctamente el vector que contiene los objetos del tipo `Image`.

El método principal de la clase es el método `selFrame`, se observa que dicho método ha sido sobrecargado. En el primero se pasa por parámetro el índice de la posición del vector que contiene la imagen que se pretende seleccionar. En el segundo se pasan dos parámetros el primero hace referencia a la orientación del jugador a representar y el segundo es la inicialización de un contador que controla la postura del jugador. Este contador puede tener valores entre 0 y 2 y una vez alcanzado el valor máximo se resetea. Mediante las llamadas a éste método se genera la sensación de que el jugador está andando.

```
class Sprite {

     private int numFrames,frame;
     private int vert_hor;
     private Image[] sprites;
     private int posx, posy;
     private int i;

     public Sprite(int numFrames) {

       frame=1;
       this.numFrames=numFrames;
       sprites=new Image[numFrames+1];
       i=0;
       vert_hor=Graphics.HCENTER|Graphics.VCENTER;

     }

public void selFrame(int frameno) {
     frame=frameno;
}

public void selFrame(int frameno, int k) {

            if(k>0){
               frame=frameno+i;
               }
            i++;
            if (i>2)i=0;

}

public int frames() {
     return numFrames;
}

public void addFrame(int frameno, String path) {
     try {
       sprites[frameno]=Image.createImage(path);

     } catch (IOException e) {
       System.err.println("Can`t load the image " + path +
       ": " + e.toString());
     }
}

public void draw(Graphics g) {

g.drawImage(sprites[frame],posx,posy,Graphics.HCENTER|Graph
ics.VCENTER);
```

```
}


}
```

Anexo 3: Código del método paint() de la clase SSCanvas.

```
public void paint (Graphics g) {

     switch(mode){

     case PREINICI:
          g.setColor(255,255,255);
          g.fillRect(0,0,getWidth(),getHeight());
          g.setColor(0,0,0);
          g.drawString("Iniciando  Juego",  (getWidth()/2),
          (getHeight()/2),Graphics.BASELINE|Graphics.HCENTE
          R);
     break;

     case INICI:
          if(opIniciSelec == 3){
               IniciSelChar.draw(g);
          }else
          if(opIniciSelec == 0)
               Inici.draw(g);

     break;

     case HISTORIA:
          Fondo.setX(getWidth()/2);
          Fondo.setY(getHeight()/2);
          Fondo.draw(g);
          FramePantalla.setX(getWidth()/2);
          FramePantalla.setY(getHeight()/2);
          FramePantalla.draw(g);
     //pinto barra de vida y de magia
        BarraVM.selFrame(1);
        BarraVM.setY(147);
        int x;
          for(x=0; x<vida/6; x++){
               BarraVM.setX(45+(x*5));
               BarraVM.draw(g);
          }
        BarraVM.selFrame(2);
        BarraVM.setY(167);
        int y;
          for(y=0; y<magia/6; y++){
               BarraVM.setX(45+(y*5));
               BarraVM.draw(g);
```

```
     }

//evaluo si m'he unit a un jugador rival o encara soc
//independent
//y pinto en cada cas el ninot que toca.

if(idmaster != -1){
     try{
          unio.ActualitzaRival(0,
          rivals.rival[idmaster].getPosX()-10,
          rivals.rival[idmaster].getPosY()-10);

          unio.rival[0].ninotRival.draw(g);
               miSprite.setX(unio.rival[0].getPosX());
               miSprite.setY(unio.rival[0].getPosY());
     }catch (Exception e){
          System.out.println("problema per accedir a
          unio");
          }
}else{
     miSprite.setX(miSprite.getX() );
     miSprite.setY(miSprite.getY() );
     miSprite.draw(g);
}
//pinto rivals
 int i,Riv;
 Riv=0;
 for(i=0;i<rivals.getNumRivals();i++){

     if(rivals.rival[i].getHab()==numFondo){

               Riv ++;
               idJugObj=i; // això s'haura de canviar

               rivals.rival[i].ninotRival.draw(g);
               if(!selector.ExisteId(i)){
               selector.AddId(i);
               }
     }else{
          if(selector.ExisteId(i)){
               selector.RemoveId(i);
          }
     }
}

Atacar = Riv;
selector.marca.setX(rivals.rival[selector.getIdSelec()
].getPosX());
selector.marca.setY(rivals.rival[selector.getIdSelec()
].getPosY());
selector.marca.selFrame(0);
```

```
selector.marca.draw(g);
}
 if(menHist == 1){
          menuHistoria.draw(g);
 }

 String identificador = new String("id = "+ id);
 g.setColor(0,0,0);
 g.drawString(identificador,miSprite.getX(),
miSprite.getY()-30,
 Graphics.TOP|Graphics.HCENTER);

break;


case BATALLA:
   if(batalla.getNumJugadors() <= 1){
          mode = HISTORIA;


    //pintar fons
     FondoBatalla.setX(getWidth()/2);
     FondoBatalla.setY(getHeight()/2);
     FondoBatalla.draw(g);
     FramePantalla.setX(getWidth()/2);
     FramePantalla.setY(getHeight()/2);
     FramePantalla.draw(g);

  //si he cridat el menu, el pinto

     if(menBat==1){
          menuBatalla.draw(g);
     }

     if(menAtac == 1){
          menuAtac.draw(g);
     }

     if(menMag == 1){
          menuMagia.draw(g);
     }

     //pinto els jugadors de la batalla.

     for(int n =0;n<batalla.getNumJugadors(); n++){

     batalla.Jugadors.rival[n].ninotRival.draw(g);
     }
```

```
        for(int j=0;j<batalla.getNumJugadors();j++){


        if (batalla.Jugadors.rival[j].getAssolitX() == 1
        && batalla.Jugadors.rival[j].getRetorn()==0)
        {
        switch (batalla.Jugadors.rival[j].getArma()){
                  case 0:
                   batalla.animacioGel.draw(g);
                  break;

                  case 1:
                   batalla.animacioEspasa.draw(g);
                  break;

                  case 2:
                       batalla.animacioPuny.draw(g);
                  break;
                  case 3:
                       batalla.animacioFoc.draw(g);
                  break;

                  case 4:
                       batalla.animacioBomba.draw(g);
                  break;

             }
             vida = dades[5];
          }
       }

        //pinto des barres de vida y magia.
        BarraVM.selFrame(1);
        BarraVM.setY(147);
        int xa;
        for(xa=0; xa<vida/6; xa++){
             BarraVM.setX(45+(xa*5));
             BarraVM.draw(g);
        }
        BarraVM.selFrame(2);
        BarraVM.setY(167);
        int ya;
        for(ya=0; ya<magia/6; ya++){
             BarraVM.setX(45+(ya*5));
             BarraVM.draw(g);
        }
    if(selector.isActiu()){
    selector.marca.setX(batalla.Jugadors.rival[batalla.tra
    nslate(selector.getIdSelec())].getPosX());
```

```
     selector.marca.setY(batalla.Jugadors.rival[batalla.tra
     nslate(selector.getIdSelec())].getPosY());
               selector.marca.selFrame(0);
               selector.marca.draw(g);
               }
          }
          break;

        }
     }
```

Anexo 4: método ActulizarFondo() de SSCanvas

```
public int ActualizarFondo(int n){
     // n hace referencia al fondo que se abandona
     if(costat != 0){
          switch (n){
               case 1:
                    if (costat == ESQUERRA){}
                    if (costat == DRETA){
                         n = 2;
                         miSprite.setX(7);
                         }
                    if (costat ==  DALT){}
                    if (costat == BAIX){}
               break;

               case 2:
                    if (costat == ESQUERRA){
                         n = 1;
                         miSprite.setX(getWidth()-7);
                         }
                    if (costat == DRETA){
                         n = 3;
                         miSprite.setX(7);
                         }
                    if (costat ==  DALT){}
                    if (costat == BAIX){
                         n = 5;
                         miSprite.setY(7);
                         }
               break;

               case 3:
                    if (costat == ESQUERRA){
                         n = 2;
                         miSprite.setX(getWidth()-7);
                         }
                    if (costat == DRETA){}
                    if (costat ==  DALT){}
```

```
                    if (costat == BAIX){}
                break;

                case 4:
                    if (costat == ESQUERRA){}
                    if (costat == DRETA){
                         n = 5;
                         miSprite.setX(7);
                         }
                    if (costat ==  DALT){}
                    if (costat == BAIX){}
                break;

                case 5:
                    if (costat == ESQUERRA){
                         n = 4;
                         miSprite.setX(getWidth()-7);
                         }
                    if (costat == DRETA){
                         n = 6;
                         miSprite.setX(7);
                         }
                    if (costat ==  DALT){
                         n=2;
                         miSprite.setY(getHeight()-46);
                         }
                    if (costat == BAIX){}
                break;

                case 6:
                    if (costat == ESQUERRA){
                         n = 5;
                         miSprite.setX(getWidth()-7);
                         }
                    if (costat == DRETA){}
                    if (costat ==  DALT){}
                    if (costat == BAIX){}
                break;
            }

        }
        //System.out.println("costat: " + costat);
        return n;
    }
```

Anexo 5: Código del método keyPressed() de la clase SSCanvas

```
protected void keyPressed(int keyCode) {

    int action = getGameAction(keyCode);
    if(!selector.isActiu() && menHist ==0){
```

```
      switch (action) {

        case FIRE:
            // Disparar
            break;
        case LEFT:
            // Mover a la izquierda
          miSprite.selFrame(10);
          k=1;
          incrX = -5;
            break;
        case RIGHT:
            // Mover a la derecha
          miSprite.selFrame(7);
          k=2;
               incrX =  + 5;

            break;
        case UP:
            // Mover hacia arriba
          miSprite.selFrame(4);
          k=3;
          incrY =  - 5;

            break;
        case DOWN:
            // Mover hacia abajo

             incrY =  + 5;
             k=4;
             miSprite.selFrame(1);

            break;
    }
  }
}
```

Anexo 6: Código comentado de la clase Menú

Esta clase permite añadir fácilmente nuevas opciones a los menús que se representan por pantalla mediante el método `addOpcio(String textOpcio)` al que únicamente hay que pasarle por parámetro el texto que representa a la opción para que se añada a la lista de opciones. Es desde el control de eventos de teclado donde se controla la lógica de selección, pero la creación de la imagen a mostrar por pantalla se gestiona totalmente desde ésta clase. Menu controla también la opción resaltada en todo momento y la navegación por las diferentes opciones mediante los métodos

```
public class Menu {
     public Sprite Opcions;
     public Image fons = null;
```

```
private int posX = 90;
private int numOpc, opcResalt;
public String[] llistaOpcions = new String[8];

public Menu(){
inicialitzaLlista();
}

public Menu(String pathFons){

try {
     fons = Image.createImage(pathFons);
} catch (IOException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
}

inicialitzaLlista();

}

public Menu(int n, String nomImatgeOpcions, String
pathImatgeFons ){
     opcResalt = 1;
     numOpc = n;
     Opcions = new Sprite(1+(2*n));
     int i;
     Opcions.addFrame(0,pathImatgeFons);
     String path;

     for (i=1;i<(2*numOpc)+1;i++){
          path = nomImatgeOpcions;
          Opcions.addFrame(i,path.concat(i+".png"));
     }

}
public void inicialitzaLlista(){

     opcResalt = 0;
     numOpc = 0;
     for (int i=0;i<8; i++){
     llistaOpcions[i] = "";
     }

}

public void addOpcio(String textOpcio){
     llistaOpcions[numOpc] = textOpcio;
     numOpc++;

}
```

```
    public void draw(Graphics g){
          int i,offset, y;
          y = 90 - ((numOpc/2)*10);
          offset=0;
          if ( fons != null){
          g.drawImage(fons,90,88,Graphics.HCENTER|Graphics
          .VCENTER);
                    }
          g.setColor(255,255,255);
          for(i=0;i<numOpc;i++){
              offset = i*10;
              if(i== opcResalt){
                   g.setColor(255,0,0);
                   g.drawString(llistaOpcions[i],posX,y+of
                   fset,Graphics.BASELINE|Graphics.HCENTER
                   );
                   g.setColor(255,255,255);
              }else{

              g.drawString(llistaOpcions[i],posX,y+offset,
          Graphics.BASELINE|Graphics.HCENTER);
                         }
                   }
    }

    public void Puja(){
    if (opcResalt>0)
         opcResalt--;
    else opcResalt = numOpc-1;
    }

    public void Baixa(){
    if (opcResalt<numOpc-1)
         opcResalt++;
    else opcResalt = 0;
    }
}
```

Anexo 7: Código del método run() de la clase SSCanvas.

```
public void run(){
Iniciar();
comunicacio.iniciaDades();

while(true){
  switch (mode){
     case INICI:
       dades = comunicacio.getDades();
       if(dades[0]==1){
```

```
          miSprite.setX(dades[2]);
          miSprite.setY(dades[3]);
         numFondo = dades[4];
         dades[0]=-1;
       }
     break;

     case HISTORIA:
          if(idmaster==-1){
               ActualizarJugador();
               numFondo = ActualizarFondo(numFondo);
               Fondo.selFrame(numFondo);
          //crear i enviar paquet
               if(incrX != 0 || incrY !=0){
                    comunicacio.setX(miSprite.getX());
                    comunicacio.setY(miSprite.getY());
                    comunicacio.setHab(numFondo);

                    try {

                    comunicacio.enviarAccion(comunicacio.se
                    nder,2);
                         } catch (IOException e1) {
                         // TODO Auto-generated catch block
                              e1.printStackTrace();
                         }
               }
          }

          dades = comunicacio.getDades();
          if(dades[0]==2){
               if(!rivals.rival[dades[4]].isFramesCarregats
               () ){
               rivals.carregaFrames(dades[4],dades[5]);
               }

               rivals.ActualitzaRival
                    (dades[4],dades[1],dades[2]);
               rivals.rival[dades[4]].setHab(dades[3]);
                    rivals.rival[dades[4]].setId(dades[4]);
                    if(idmaster != -1 && idmaster ==
                    dades[4]){
                    //si el paquet rebut pertany al meu
                    //master
                    // actualitzo el ninot d'unio amb les
                    //dades del master
                    unio.ActualitzaRival(0
                    ,rivals.rival[idmaster].getPosX()-
                    10,rivals.rival[idmaster].getPosY()-
                    10);
```

```
                unio.rival[0].setHab(
                rivals.rival[idmaster].getHab());
                numFondo = unio.rival[0].getHab();
                Fondo.selFrame(numFondo);

                //actualitzo comunicacio amb les dades
                //del ninot d'unio
                comunicacio.setX(
                unio.rival[0].getPosX());

                comunicacio.setY(
                unio.rival[0].getPosY());

                comunicacio.setHab(
                unio.rival[0].getHab());

                //envio paquet d'accio al servidor.

                try {
                     comunicacio.enviarAccion(
                     comunicacio.sender,2);
                     } catch (IOException e1) {
                     // TODO Auto-generated catch block
                          e1.printStackTrace();
                     }

                }
                dades[0]=-1;

           }

           //escolto si hi ha ordre de batalla

      if(dades[0]==3){
        switch(dades[1]){

           case 0:
                ode = BATALLA;
                batalla = new Batalla(dades);

                selector.IniciaSelector();
                for(int x=0;x<batalla.getNumJugadors();
                x++){
                     if(batalla.Jugadors.rival[x].getId
                     ()!= comunicacio.getId()){
                     selector.AddId(
                     batalla.Jugadors.rival[x].getId())
                     ;
                     }
                }
                batalla.setParar(0);
```

```
                for(int b =0;
                b<batalla.getNumJugadors();b++){

                if(batalla.Jugadors.rival[b].getId()==
                comunicacio.getId()){
                     batalla.Jugadors.carregaFrames(b,
                     comunicacio.getPersonatge());
                }



                for(int p = 0; p<
                rivals.getNumRivals();p++){
                if(batalla.Jugadors.rival[b].getId()==
                rivals.rival[p].getId()){
                     batalla.Jugadors.carregaFrames(b,
                     rivals.rival[p].getPersonatge());
                     }
                }
                }

          Thread miBatalla = new Thread (batalla);
          miBatalla.start();

                dades[1]=-1;
          break;



          }
     }
     if(dades[0]== 4){
      switch(dades[1]){
          case 0:
                unio.carregaFrames(0,
                comunicacio.getPersonatge());
                if(dades[2]!= id){
                     idmaster = dades[2];
                     unio.ActualitzaRival(0,
                     rivals.rival[idmaster].getPosX()-
                     10,rivals.rival[idmaster].getPosY(
                     )-10);
                     unio.rival[0].setHab(
                     rivals.rival[idmaster].getHab());
                     numFondo = unio.rival[0].getHab();

                     comunicacio.setX(
                     unio.rival[0].getPosX());
                     comunicacio.setY(
                     unio.rival[0].getPosY());
```

```
                    comunicacio.setHab(
                    unio.rival[0].getHab());

                    try {
                         comunicacio.enviarAccion(
                         comunicacio.sender,2);
                         } catch (IOException e1) {
                         //TODO Auto-generated catch
                         block
                              e1.printStackTrace();
                              }
                         }else{
                              if(dades[2]== id)
                                   idslave = 1;
                         }

                    break;
                    case 1:
                         if(dades[2] != id){
                         idmaster= -1;
                         }else {
                              if (dades[2] == id &&
                              idslave == 1){
                                   idslave = -1;
                              }
                         }
                    break;
               }
               dades[0] = -1;

          }

     break;

     case BATALLA:
          dades = comunicacio.getDades();
          if (dades[1]==1){

               batalla.actualitzaBatalla(dades);

               if(dades[9]!=-1){
                    if(dades[9] ==
                    comunicacio.getId()){

                         mort = 1;
                    }else{

                         selector.RemoveId(dades[9]);
                         rivals.rival[
                         dades[9]].setHab(-1);
```

```
                                }

                        }
                        dades[1]=-1;

                }
                if(dades[1]==9){
                        selector.RemoveId(dades[2]);
                        batalla.eliminaRival(dades[2]);
                        dades[1]=-1;
                }
                break;

        }

        if(mort==1 && batalla.getFiAnimacio()==1){
                        mode = INICI;

        }


    try {
     Thread.sleep(180); //a 180 funciona be
     } catch (InterruptedException e) {
          System.out.println(e.toString());
     }

    repaint();
    serviceRepaints();

     }


}
```

Anexo 8: Código del método envia() de la clase Enviar

```
public void envia(int tipo){

     switch (tipo){
          case 1:
          if (tipo ==1){

          String dades = new String(tipo+
          "/"+id+"/"+x+"/"+y+"/"+hab+"/"+"/");

          byte[] msg = dades.getBytes();
          paquetSort = new
          DatagramPacket(msg,msg.length,ip,port);
               try {
```

```
                sortida.send(paquetSort);
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

        break;

        case 2:

        String dadesH = new String
        (2+"/"+llista[id].getX()+"/"+
        llista[id].getY()+"/"+llista[id].getHab()+"/"+id+
        "/"+personatge+"/"+"/");

        byte[] msgH = dadesH.getBytes();
        paquetSortH = new
        DatagramPacket(msgH,msgH.length,ip,port);

        try {
            sortida.send(paquetSortH);
            } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("falla el try del case 2,
            no envia paquet sortida");
            }

        break;

        // cas de confirmació d'ordre de batalla

        case 3:

            String dadesOB = new String
            (3+"/"+0+"/"+numJug+"/"+id+"/"+idObj+"/"+"/"
            );
            byte[] msgOB = dadesOB.getBytes();
            paquetSortOB = new
            DatagramPacket(msgOB,msgOB.length,ip,port);

            try{
                sortida.send(paquetSortOB);
            }catch (IOException e ){
                e.printStackTrace();
                System.err.println("imposible enviar
                orde de batalla");
            }
```

```
break;


case 30:

     String llista = new String();
     llista="";
     for(int i=0; i<numJug;i++){
          llista=llista.concat("/"+llistaId[i]);
     }
     String dadesOB2 = new String
     (3+"/"+0+"/"+numJug+llista+"/"+"/");
     byte[] msgOB2 = dadesOB2.getBytes();
     paquetSortOB2 = new
     DatagramPacket(msgOB2,msgOB2.length,ip,port)
     ;

     try{
          sortida.send(paquetSortOB2);
     }catch (IOException e ){
          e.printStackTrace();
          System.err.println("imposible enviar
          orde de batalla multiple");
     }
break;

case 31:
     String dadesOA = new String
     (3+"/"+1+"/"+idObj+"/"+id+"/"+arma+"/"+HP+"/
     "+MP+"/"+Atac+"/"+Def+"/"+mort+"/"+"/");
     byte[] msgOA = dadesOA.getBytes();
     paquetSortOA = new
     DatagramPacket(msgOA,msgOA.length,ip,port);

     try{
          sortida.send(paquetSortOA);
     }catch (IOException e){
          e.printStackTrace();
          System.err.println("imposible    enviar
          ordre atac");
     }
break;

case 39:
     String dadesFB = new String
     (3+"/"+9+"/"+id+"/"+"/");

     System.out.println("envio paquet fora
     batalla " + dadesFB);
     byte[] msgFB = dadesFB.getBytes();
```

```
                paquetSortFB = new
                DatagramPacket(msgFB,msgFB.length, ip,
                port);

                try{
                     sortida.send(paquetSortFB);
                    }catch (IOException e){
                     e.printStackTrace();
                     System.err.println("imposible enviar
                     ordre Fora Batalla");
                }

           break;

           case 4:
                String dadesOU = new String
                (4+"/"+solicitud+"/"+id+"/"+"/");
                byte[] msgOU = dadesOU.getBytes();
                paquetSortOU = new DatagramPacket (msgOU,
                msgOU.length, ip, port);

                try{
                     sortida.send(paquetSortOU);
                }catch (IOException e){
                     e.printStackTrace();
                     System.err.println("imposible    enviar
                     ordre Unio");
                }
           break;
     }
}
```